编译原理笔记

哈工大编译原理期末复习（完整版）-CSDN博客

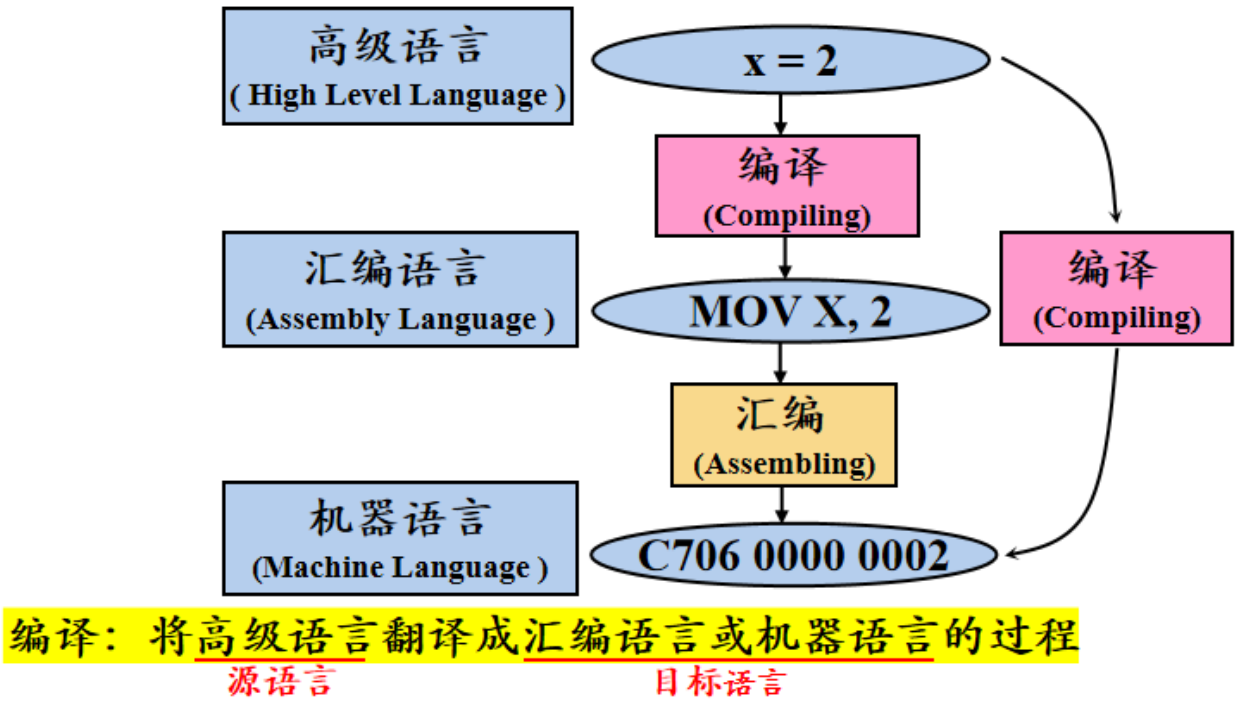
可能是全网最全的哈工大编译原理资料分享 - 知乎 (zhihu.com)

编译原理习题_花月诗人的博客-CSDN博客

⚠ 习题整理

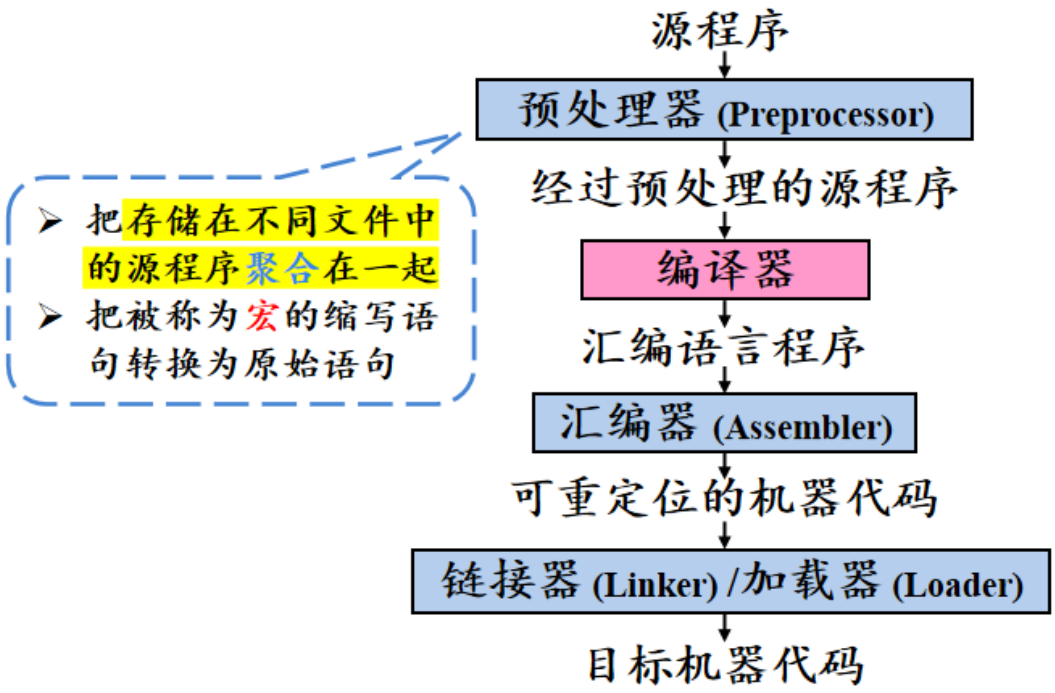
第一章 绪论

1.1 编译



- **编译：源语言 → 目标语言**
  - 目标语言可以是汇编语言，也可以是机器语言

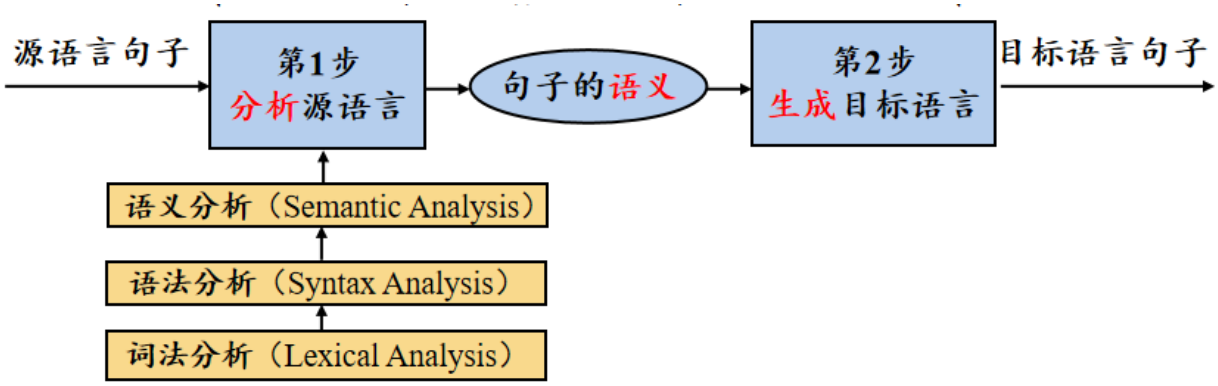
编译器在语言处理系统中的位置

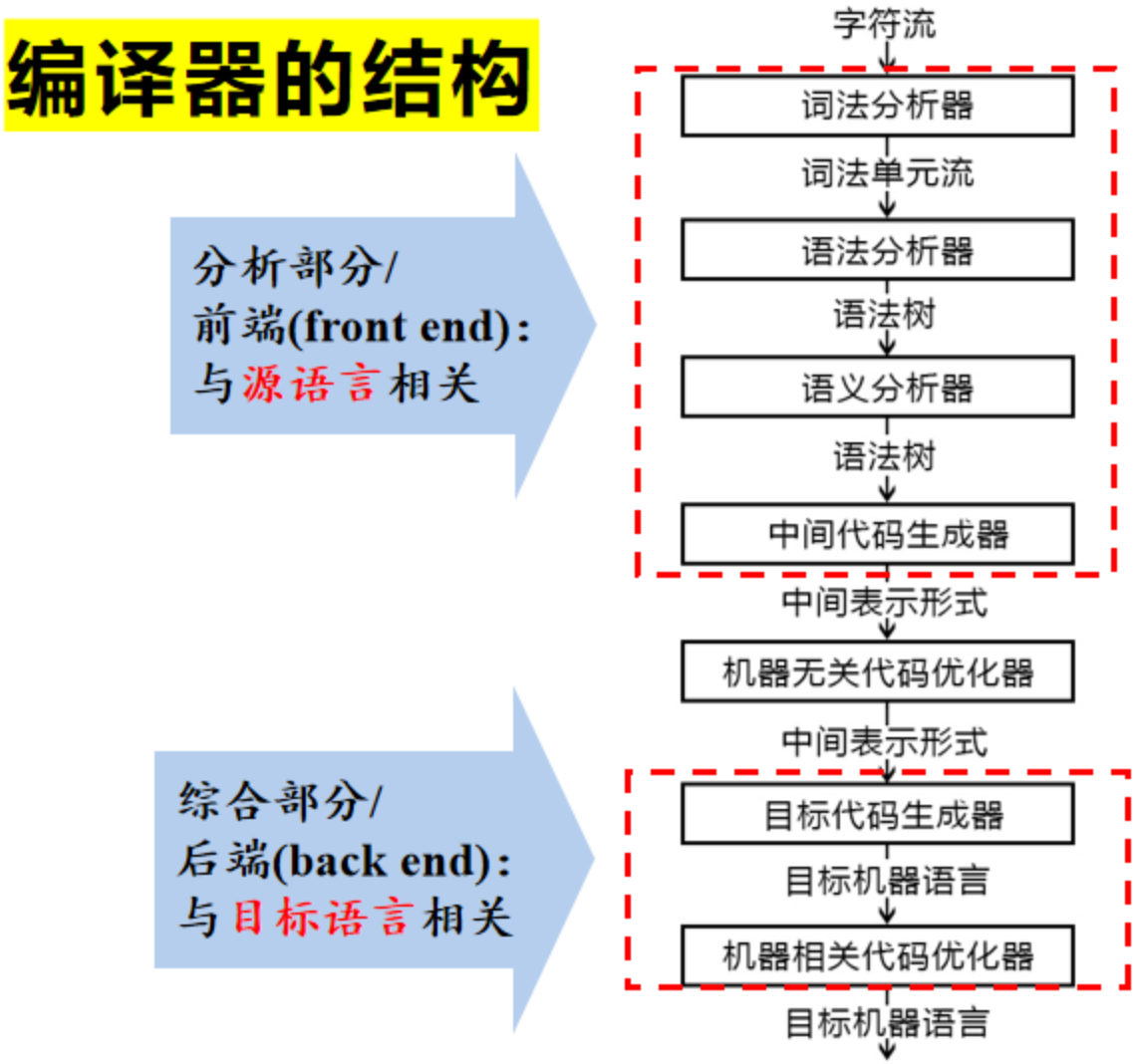


- 可重定位：在内存中存放的起始位置L不是固定的
- 加载器和链接器不是一个东西
  - 加载器：
    - 修改可重定位地址；
      - 起始位置 + 相对地址 = 绝对地址
    - 将修改后的指令和数据放到内存中适当的位置
  - 链接器：
    - 将多个可重定位的机器代码文件（包括库文件）连接到一起
    - 解决外部内存地址问题

1.2 编译系统的结构

- 编译程序是一种翻译程序





- 注：中间进行了两次优化：一次是**机器无关**代码优化，一次是**机器相关**代码优化
- **词法分析 → 语法分析 → 语义分析**

1.2.1 词法分析器（单词）

词法分析/扫描(*Scanning*)

➢ 词法分析的主要任务

从左向右逐行扫描源程序的字符，识别出各个单词，确定单词的类型。

将识别出的单词转换成统一的机内表示——词法单元(token)形式

token：＜种别码，属性值＞

	单词类型	种别	种别码
1	关键字	program、if、else、then、…	一词一码
2	标识符	变量名、数组名、记录名、过程名、…	多词一码
3	常量	整型、浮点型、字符型、布尔型、…	一型一码
4	运算符	算术（+ - * / ++ --） 关系（> < == != >= <=） 逻辑（& | ~）	一词一码 或 一型一码
5	界限符	; ( ) = { } …	一词一码

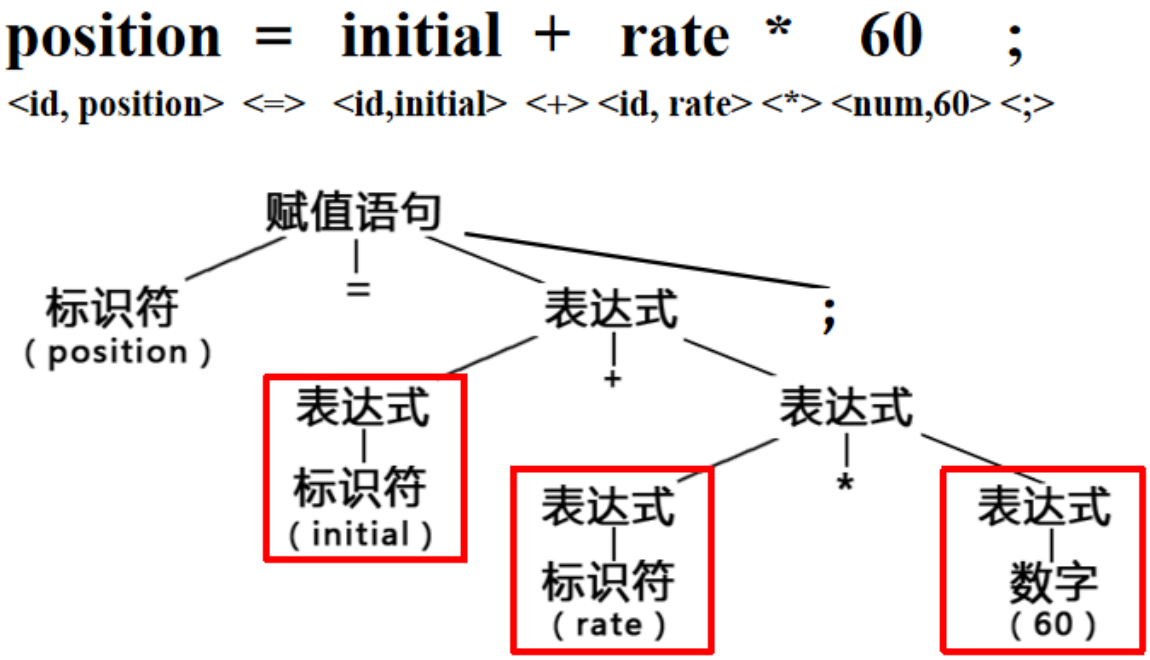
- 词法分析又叫扫描
- **token：<种别码，属性值>**

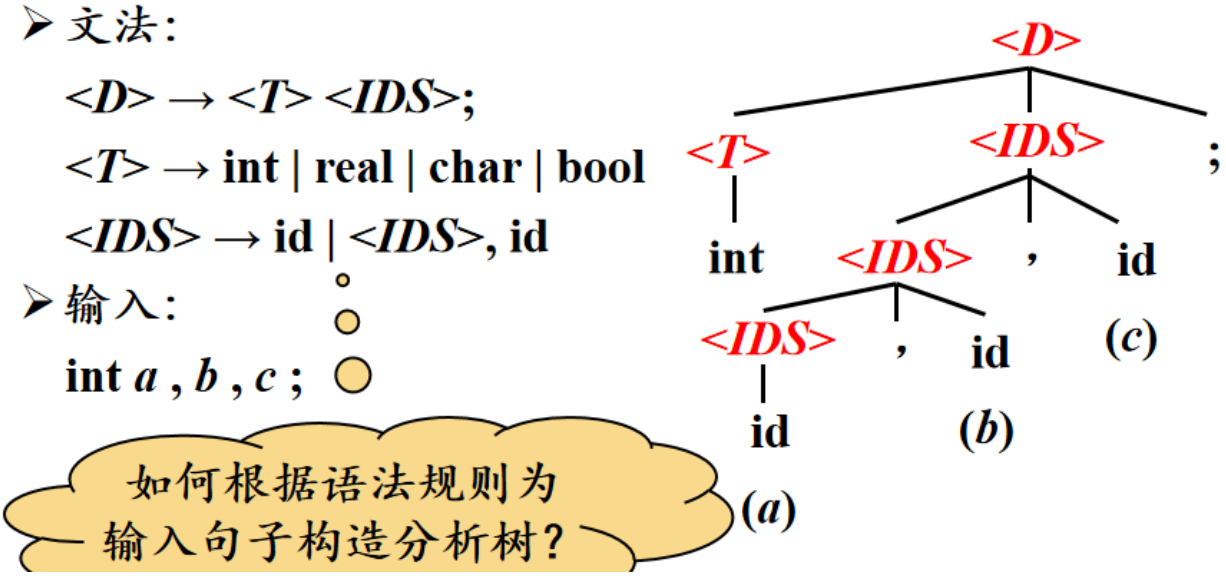
- 输入　while(value!=100){num++;}
- 输出

1	while	<	WHILE	-	>
2	(	<	SLP	-	>
3	value	<	IDN	value	>
4	!=	<	NE	-	>
5	100	<	CONST	100	>
6	)	<	SRP	-	>
7	{	<	LP	-	>
8	num	<	IDN	num	>
9	++	<	INC	-	>
10	;	<	SEMI	-	>
11	}	<	RP	-	>

1.2.2 语法分析器（短语）

- 语法分析器(parser)从词法分析器输出的token序列中识别出各类短语，并构造语法分析树(parse tree)
  - 语法分析树描述了句子的语法结构

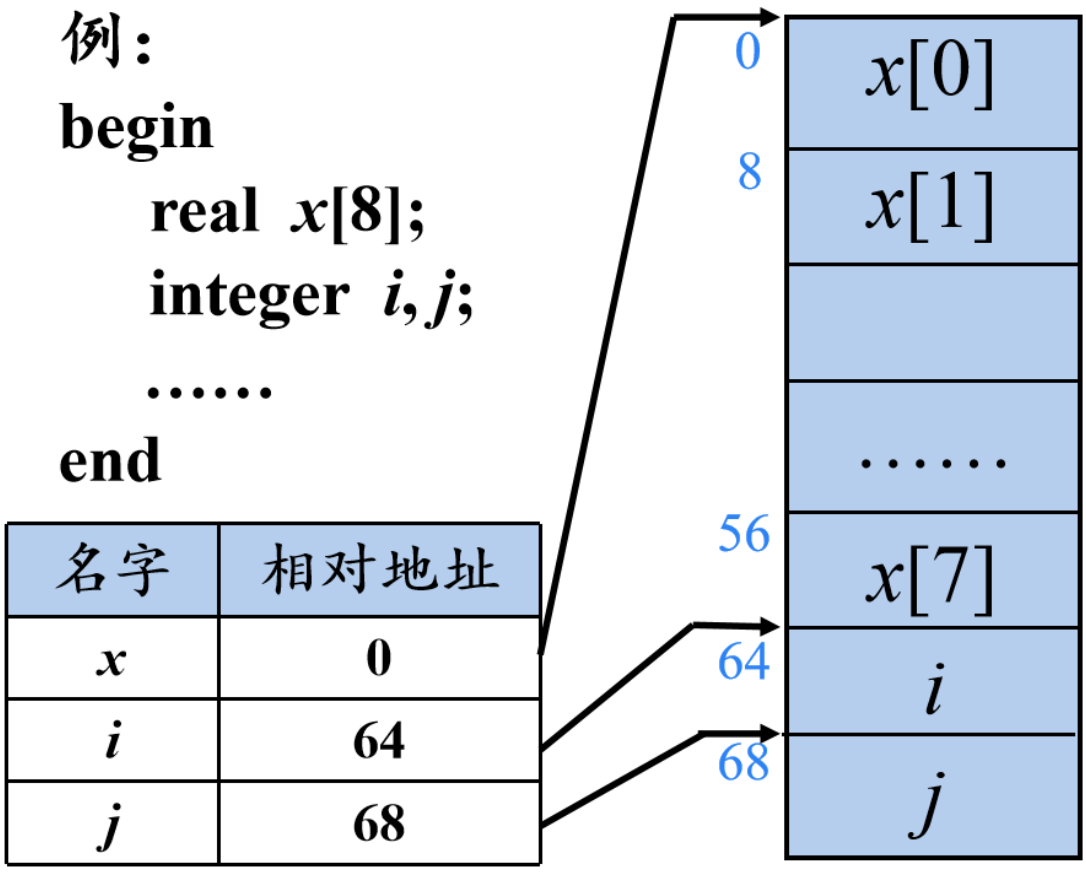




1.2.3 语义分析器

语义分析的**主要任务：收集标识符的属性信息+语义检查**

- **收集标识符的属性信息**
  - 种属（Kind）：简单变量、复合变量...
  - 类型（Type）：整型、实型、字符型...
  - 存储位置、长度



  - 值
  - 作用域
  - 参数和返回值信息

符号表是用于存放标识符的属性信息的数据结构

符号表(Symbol Table)

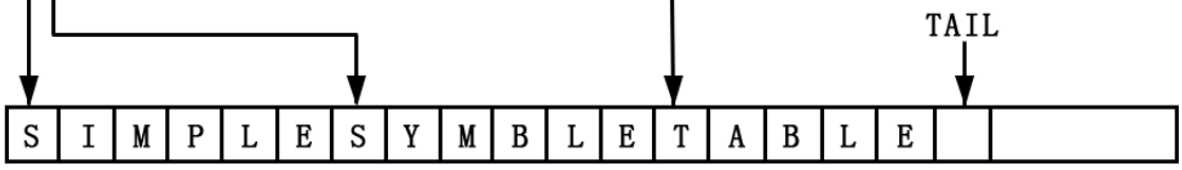
NAME	TYPE	KIND	VAL	ADDR
SIMPLE	整	简变		
SYMBLE	实	数组		
TABLE	字符	常数		
⋮	⋮	⋮	⋮	⋮

字符串表

首地址+长度

符号表(Symbol Table)

NAME		TYPE	KIND	VAL	ADDR
	6	整	简变		
	6	实	数组		
	5	字符	常数		
⋮	⋮	⋮	⋮	⋮	⋮



- T 1.2

➢ 符号表中NAME字段为什么要设计字符串表这样一种数据结构？而不是把标识符对应的字符串直接存放到NAME字段？

若把标识符对应的字符串直接存放到NAME字段，
考虑：定长OR变长

定长：太长的→存不下，溢出；
太短的→大量空间空闲，造成空间浪费；
变长：查询效率低下。
其他考量因素：便于字符串的统一管理

- **语义检查**
  - 变量（包括数组、指针、结构体）或过程未经声明就使用
  - 变量（包括数组、指针、结构体）或过程名重复声明
  - 运算分量类型不匹配
  - 操作符与操作数之间的类型不匹配

1.2.4 中间代码生成器

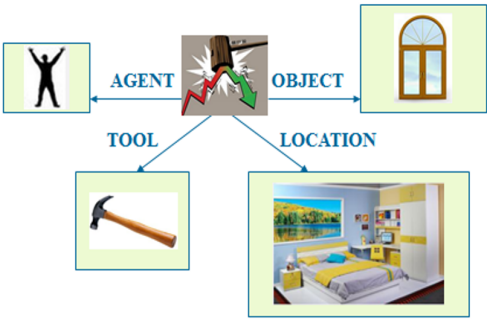
- 常用的中间表示形式
  - 三地址码
  - 语法结构树/语法树
- **三地址码**
  - 三地址码由类似于汇编语言的指令序列组成，每个指令**最多有三个操作数**
  - **地址可以具有如下形式之一**

    - 源程序中的名字 (name)
    - 常量 (constant)
    - 编译器生成的临时变量(temporary)
  - **三地址指令的表示**
    - 四元式 (Quadruples)

      (op, arg1, arg2, **result**)
    - 三元式 (Triples)

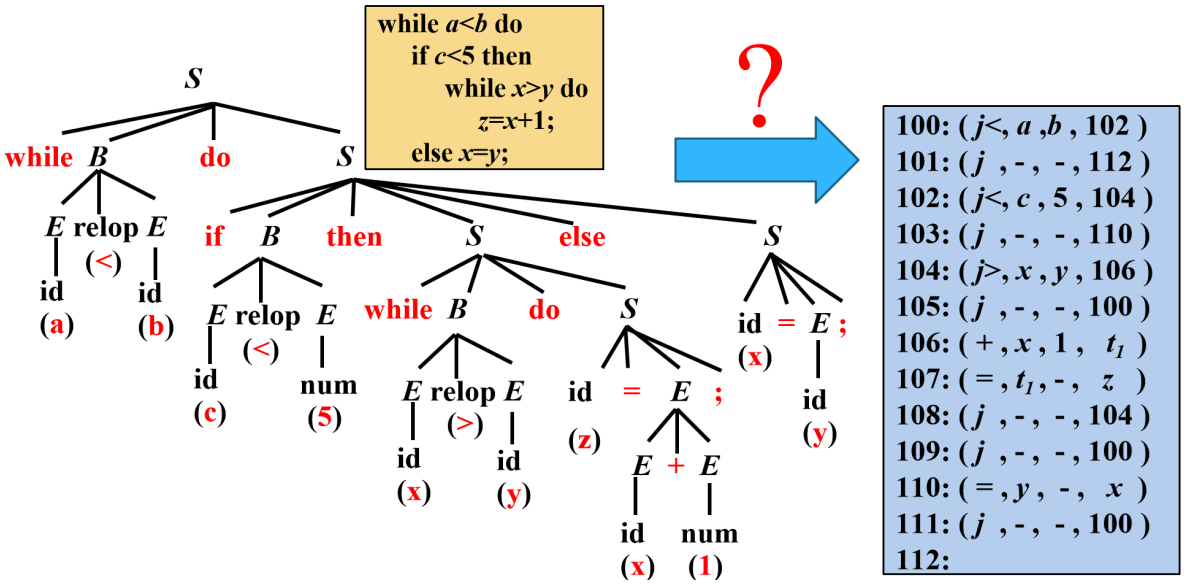
      (op, arg1, agr2)
    - 间接三元式 (Indirect triples)
  - 三地址指令的**四元式表示**

➢ $x = y$ op z	(op , y , z , x)
➢ $x =$ op y	(op , y , _ , x)
➢ $x = y$	(= , y , _ , x)
➢ if x relop y goto n	(relop , x , y , n)
➢ goto n	(goto , _ , _ , n)
➢ param x	(param, _ , _ , x)
$y =$call p, n	(call , p , n , y)
return x	(return, _ , _ , x)
➢ $y = x[i]$	(=[] , x , i , y)
$x[i] = y$	([]= , y , x , i)
➢ $x = \&y$	(& , y , _ , x)
$x = *y$	(=* , y , _ , x)
$*x = y$	(*= , y , _ , x)



三地址指令序列唯一确定了运算完成的顺序

- **中间代码生成的例子**



1.2.5 目标代码生成器

- 目标代码生成以源程序的**中间表示形式**作为**输入**，并把它**映射到目标语言**
- 目标代码生成的一个重要任务是为程序中使用的变量**合理分配寄存器**

1.2.6 代码优化（两个优化器）

- 机器无关代码优化器 & 机器相关代码优化器
- 为改进代码所进行的**等价程序变换**，使其**运行得更快**一些、**占用空间更少**一些，或者二者兼顾

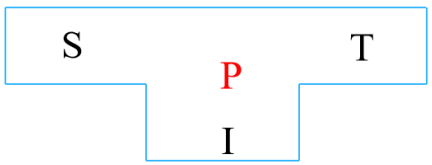
1.3 编译程序的生成

- 1970年以前，几乎所有的编译程序都是用机器语言编写的
  - 优点：更好地发挥硬件系统的效率
  - 缺点：可读性、可靠性、可维护性、编制效率差
- 1980年以后，通常用高级语言来编写编译程序（自展技术）
- 编译器的移植
  - 有时也称为交叉编译，是指将一台机器上运行的编译器进行处理，构造出在另一台机器上可以运行的编译器

- **编译器的自动生成**
  - LEX： 词法分析程序生成器
  - YACC：语法分析程序生成器

- **编译器的T型图**

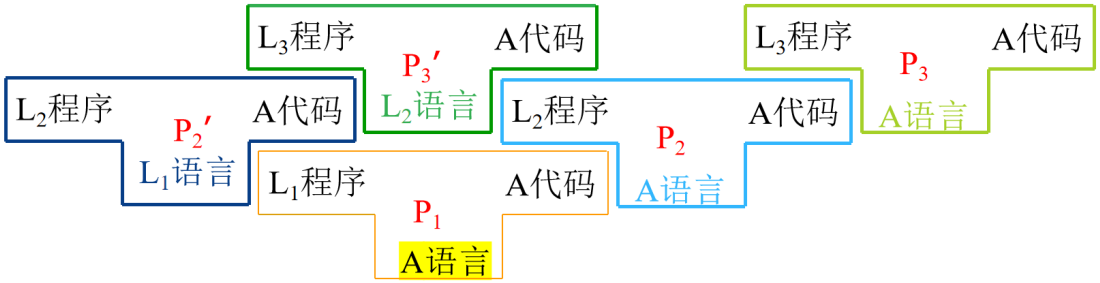
  - **P：编译器（程序）**
    - **I：实现语言**
    - **S：输入的源语言程序**
    - **T：输出的可执行的目标程序**



注意：
（1）I表示的是语言，而S和T表示的是程序
（2）T形图的上端体现了编译器的功能，即从哪种语言到哪种语言的翻译
（3）T对应于某机器语言

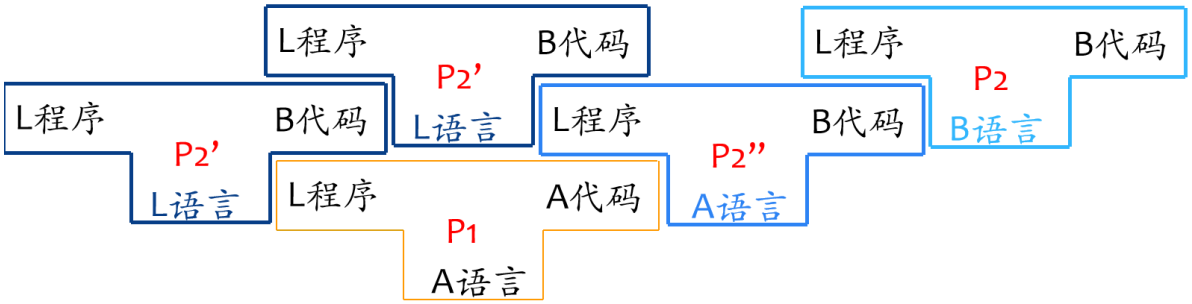
- **自展（在同一台机器上实现不同语言的编译器）**
  - 用高级语言来编写编译程序

➢ 给定P_1： A机器上运行的高级语言L_1的编译器
➢ 构造P_2： A机器上运行的高级语言L_2的编译器
➢ 构造P_3： A机器上运行的高级语言L_3的编译器



- **编译器的移植（不同机器上同一个语言的编译器）**

➢ 给定P_1： A机器上运行的高级语言L的编译器
➢ 构造P_2： B机器上运行的高级语言L的编译器



第二章 语言及其文法

2.1 基本概念

- 串：有穷符号（symbol） 序列
- 串的连接：xy
- 串的n次幂：将n个s连接起来
- 字母表：有穷符号集合
- 字母表的乘积：笛卡尔积
- 字母表的n次幂：长度为n的符号串构成的集合
- 字母表的正闭包：长度正数的符号串构成的集合
- 字母表的克林闭包：任意符号串（长度可以为零）构成的集合

2.2 文法的定义

- **未用尖括号**括起来部分表示**语言的基本符号**
  - eg. <形容词> → little
- **尖括号**括起来部分称为**语法成分**
  - eg. <句子> → <名词短语> <动词短语>
- G = (VT , VN , P , S)
  - **VT ：终结符**（有时称为Token）集合，终结符→文法定义的语言的**基本符号**
  - **VN：非终结符**（语法变量）集合，非终结符→表示**语法成分**
  - P ：产生式集合
  - S ：开始符号（最大的语法成分）
- α→β1 | β2 | ... | βn
  - β1，β2，...，βn称为α的**候选式**

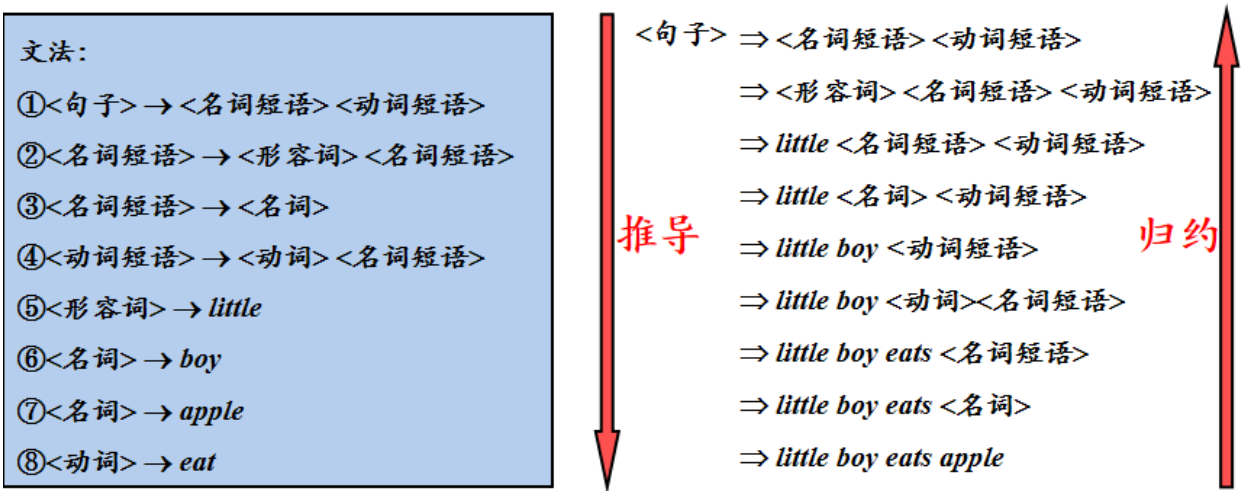
2.3 语言的定义

- **推导（派生）**：用产生式的右部替换产生式的左部（生成语言）

  - $\alpha \Rightarrow^0 \alpha$
  - $\alpha_0 \Rightarrow^n \alpha_n$ 表示 α_0 经过 n 步推导出 α_n
  - $\Rightarrow^+$ 表示“经过正数步推导”
  - $\Rightarrow^*$ 表示“经过若干（可以是0）步推导”

  α经过0步推导后还是自身

- **归约**：用产生式的左部替换产生式的右部（识别语言）



- **句型和句子**

- 如果 $S \Rightarrow^* \alpha$，$\alpha \in (V_T \cup V_N)^*$，则称$\alpha$是$G$的一个句型 (*sentential form*)
  - 一个句型中既可以包含终结符，又可以包含非终结符，也可能是空串
- 如果 $S \Rightarrow^* w$，$w \in V_T^*$，则称w是G的一个句子(*sentence*)
  - 句子是不包含非终结符的句型，只能由终结符构成

<句子>⇒ <名词短语> <动词短语>
⇒ <形容词> <名词短语> <动词短语>
⇒ *little* <名词短语> <动词短语>
⇒ *little* <名词> <动词短语>
⇒ *little boy* <动词短语>
⇒ *little boy* <动词><名词短语>
⇒ *little boy eats* <名词短语>
⇒ *little boy eats* <名词>
⇒ *little boy eats apple*

句型

句子

- **语言的形式化定义**：由文法G的开始符号S推导出的所有句子构成的集合称为文法G生成的语言，记为L(G)。

2.4 文法的分类

0型文法 (*Type-0 Grammar*)

$$\alpha \rightarrow \beta$$

- 无限制文法(*Unrestricted Grammar*)/短语结构文法 (*Phrase Structure Grammar, PSG*)
  - $\forall \alpha \rightarrow \beta \in P$，$\alpha$中至少包含1个非终结符
- 0型语言
  - 由0型文法G生成的语言$L(G)$

1型文法 (*Type-1 Grammar*)

$$\alpha \rightarrow \beta$$

- 上下文有关文法(*Context-Sensitive Grammar*, *CSG*)
  - $\forall \alpha \rightarrow \beta \in P$，$|\alpha| \leq |\beta|$
  - 产生式的一般形式：$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$ ($\beta \neq \varepsilon$)
- 上下文有关语言（1型语言）
  - 由上下文有关文法 (1型文法) G生成的语言$L(G)$

CSG中不包含ε-产生式

2型文法 (*Type-2 Grammar*)

$$\alpha \rightarrow \beta$$

- 上下文无关文法 (*Context-Free Grammar, CFG*)
  - $\forall \alpha \rightarrow \beta \in P$，$\alpha \in V_N$
  - 产生式的一般形式：$A \rightarrow \beta$
- 上下文无关语言（2型语言）
  - 由上下文无关文法 (2型文法) G生成的语言$L(G)$

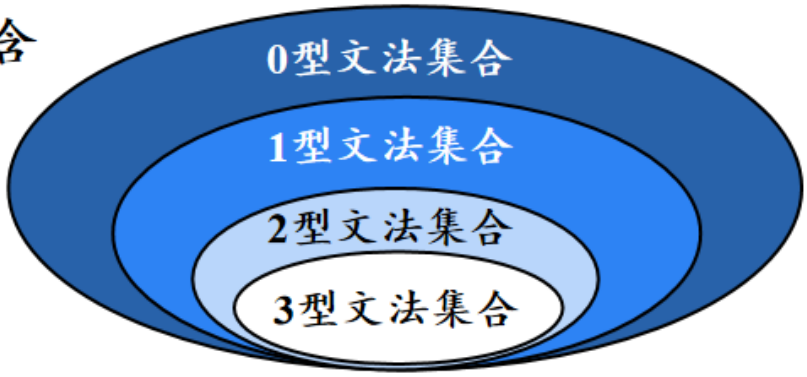
3型文法 (*Type-3 Grammar*)

$$\alpha \rightarrow \beta$$

- 正则文法 (*Regular Grammar, RG*)
  - 右线性(*Right Linear*)文法：$A \rightarrow wB$ 或 $A \rightarrow w$
  - 左线性(*Left Linear*) 文法：$A \rightarrow Bw$ 或 $A \rightarrow w$
  - 左线性文法和右线性文法都称为正则文法
- 正则语言（3型语言）
  - 由正则文法 (3型文法) G生成的语言$L(G)$

正则文法能描述程序设计语言的多数单词

- 逐级限制
  - 0型文法：α中至少包含1个非终结符
  - 1型文法（CSG）：$|\alpha| \leq |\beta|$
  - 2型文法（CFG）：$\alpha \in V_N$
  - 3型文法（RG）：$A \rightarrow wB$ 或 $A \rightarrow w$ （$A \rightarrow Bw$ 或 $A \rightarrow w$）
- 逐级包含



2.5 CFG（上下文无关文法）的语法分析树

- 分析树是**推导的图形化表示**

- 根节点的标号为文法开始符号
- 内部结点表示对一个产生式$A \rightarrow \beta$的应用，该结点的标号是此产生式左部A。该结点的子结点的标号从左到右构成了产生式的右部β
- 叶结点的标号既可以是非终结符，也可以是终结符。从左到右排列叶节点得到的符号串称为是这棵树的产出(*yield*)或边缘(*frontier*)

（句型的）短语

- 给定一个句型，其分析树中的每一棵子树的边缘称为该句型的一个短语(*phrase*)
  - 如果子树只有父子两代结点，那么这棵子树的边缘称为该句型的一个直接短语(*immediate phrase*)

文法：

① $E \rightarrow E+E$

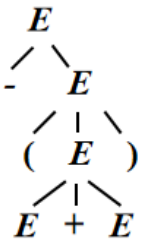
② $E \rightarrow E*E$

③ $E \rightarrow -E$

④ $E \rightarrow (E)$

⑤ $E \rightarrow id$

分析树：



短语：

- $-(E+E)$
- $(E+E)$
- $E+E$

直接短语：

- $E+E$

直接短语一定是某产生式的右部

但产生式的右部不一定是给定句型的直接短语

- **二义性文法**：如果一个文法可以为某个句子生成**多棵分析树**，则称这个文法是**二义性**的
- **二义性文法的判定**
  - 对于任意一个上下文无关文法，**不存在一个算法**，判定它是否为二义性的
  - 但能给出一组**充分条件**，满足这组充分条件的文法是**无二义性**的
    - 满足，**肯定**无二义性
    - 不满足，也**未必**就是有二义性的

第三章 词法分析

3.1 单词的描述

➢ 正则文法

例：描述标识符的正则文法

①$S \rightarrow aS' \mid bS' \mid \ldots \mid zS'$

②$S' \rightarrow aS' \mid bS' \mid \ldots \mid zS' \mid 0S' \mid 1S' \mid 2S' \mid \ldots \mid 9S' \mid \varepsilon$

➢ 正则表达式(*Regular Expression*，*RE*)

例

➢ 十进制整数的*RE*

- $(1|...|9)(0|...|9)^*|0$

➢ 八进制整数的*RE*

- $0(0|1|2|3|4|5|6|7)(0|1|2|3|4|5|6|7)^*$

➢ 十六进制整数的*RE*

- $0x(0|1|...|9|a|...|f|A|...|F)(0|...|9|a|...|f|A|...|F)^*$

是一种用来描述正则语言的 更紧凑的表示方法

- **正则文法适用于描述单词**

➢ 正则定义是具有如下形式的定义序列：

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_n \rightarrow r_n$$

给一些*RE*命名，并在之后的*RE*中像使用字母表中的符号一样使用这些名字

其中：

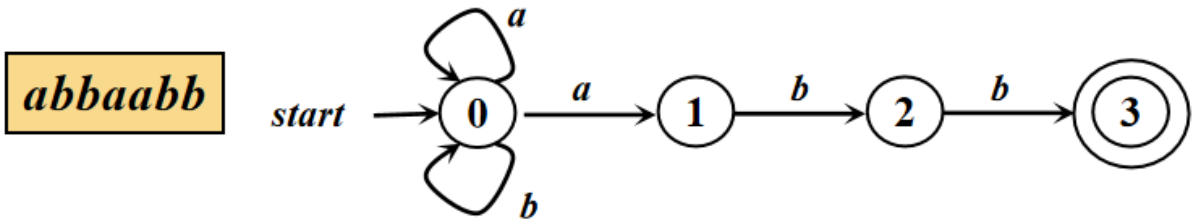
➢ 每个d_i都是一个新符号，它们都不在字母表Σ中，而且各不相同

➢ 每个r_i是字母表$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$上的正则表达式

3.2 单词的识别

3.2.1 有穷自动机FA

- 有穷自动机：系统只需要根据**当前所处的状态**和**当前面临的输入信息**就可以决定系统的后继行为

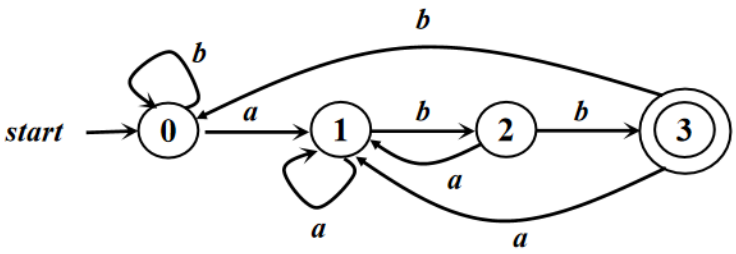


- **最长子串匹配原则：**当输入串的多个前缀与一个或多个模式匹配时，总是选择最长的前缀进行匹配

3.2.2 FA（有穷自动机）的分类

- **确定的FA（DFA）**

$$M=(S,\ \Sigma,\ \delta,\ s_0,\ F)$$



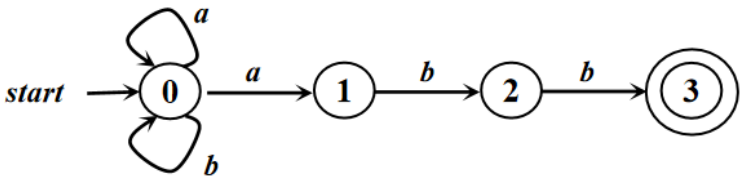
转换表

状态 \ 输入	a	b
0	1	0
1	1	2
2	1	3
3	1	0

可以用转换表表示DFA

- **非确定的FA（NFA）**

$$M=(S,\ \Sigma,\ \delta,\ s_0,\ F)$$



转换表

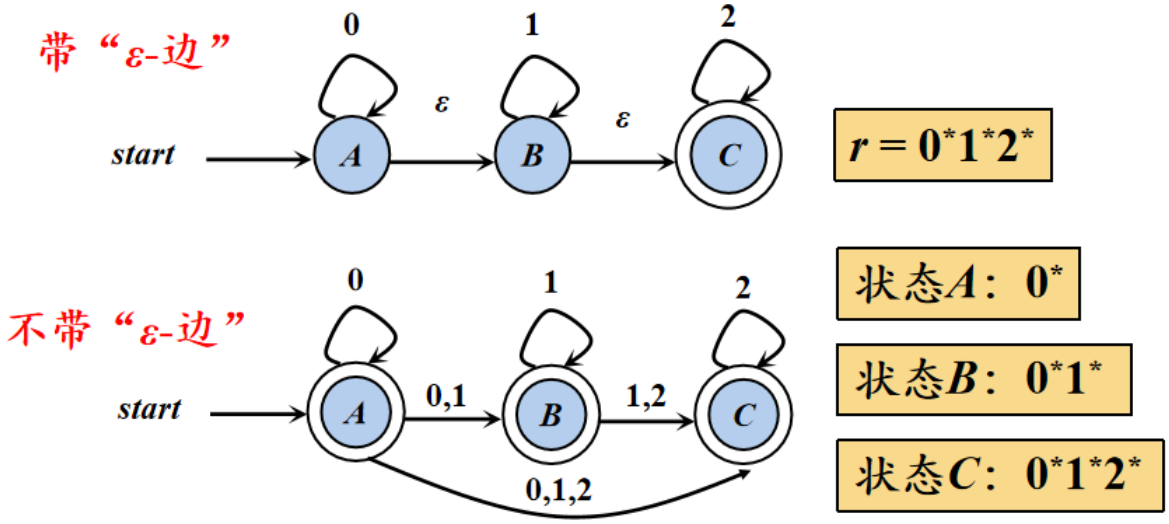
状态 \ 输入	a	b
0	{0,1}	{0}
1	Ø	{2}
2	Ø	{3}
3	Ø	Ø

如果转换函数没有给出对应于某个状态-输入对的信息，就把Ø放入相应的表项中

- **DFA和NFA具有等价性**，可以识别相同语言

正则文法 ⇔ 正则表达式 ⇔ FA

- **带有“ε-边”的 NFA**



- **带有和不带有“ε-边”的 NFA 具有等价性**

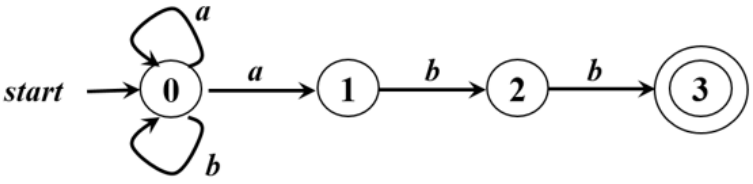
DFA的算法实现

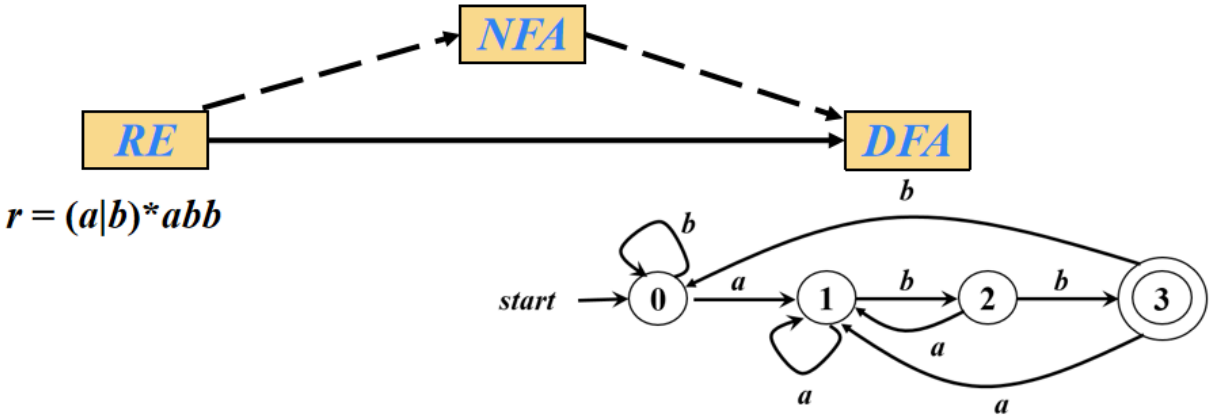
- 输入：以文件结束符eof结尾的字符串x。DFA D 的开始状态s_0，接收状态集 F，转换函数$move$。
- 输出：如果D接收x，则回答“yes”，否则回答“no”。
- 方法：将下述算法应用于输入串 x。

```
s = s0;
c = nextChar（）;
while（ c! = eof ） {
        s = move ( s , c ) ;
        c = nextChar ( ) ;
}
if (s在F中) return “yes” ;
else return “no” ;
```

- 函数nextChar( )返回输入串x的下一个符号
- 函数move(s, c)表示从状态s出发，沿着标记为c的边所能到达的状态

3.2.3 从正则表达式到有穷自动机

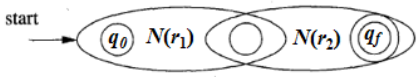




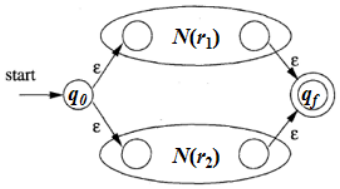
- **根据RE构造NFA**

➢假设正则表达式r_1和r_2对应的NFA分别为$N(r_1)$和$N(r_2)$

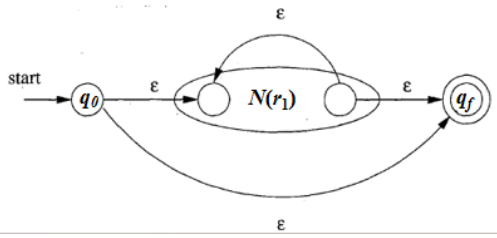
➢$r = r_1 r_2$对应的NFA



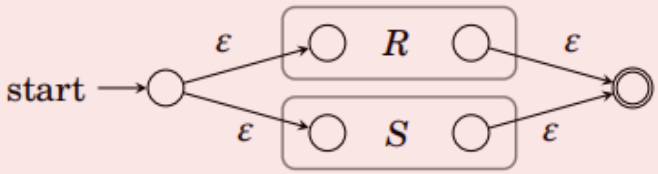
➢$r = r_1 | r_2$对应的NFA



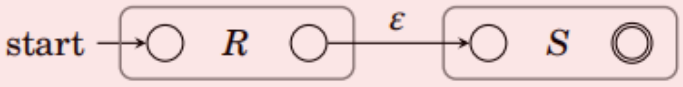
➢$r = (r_1)^*$对应的NFA



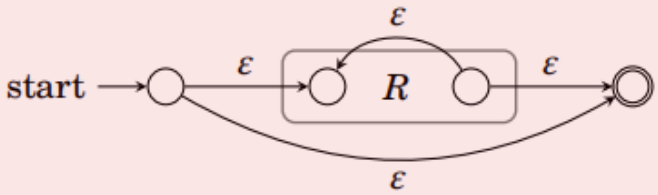
1. 对于 **r**+**s**, 有 ε-NFA:



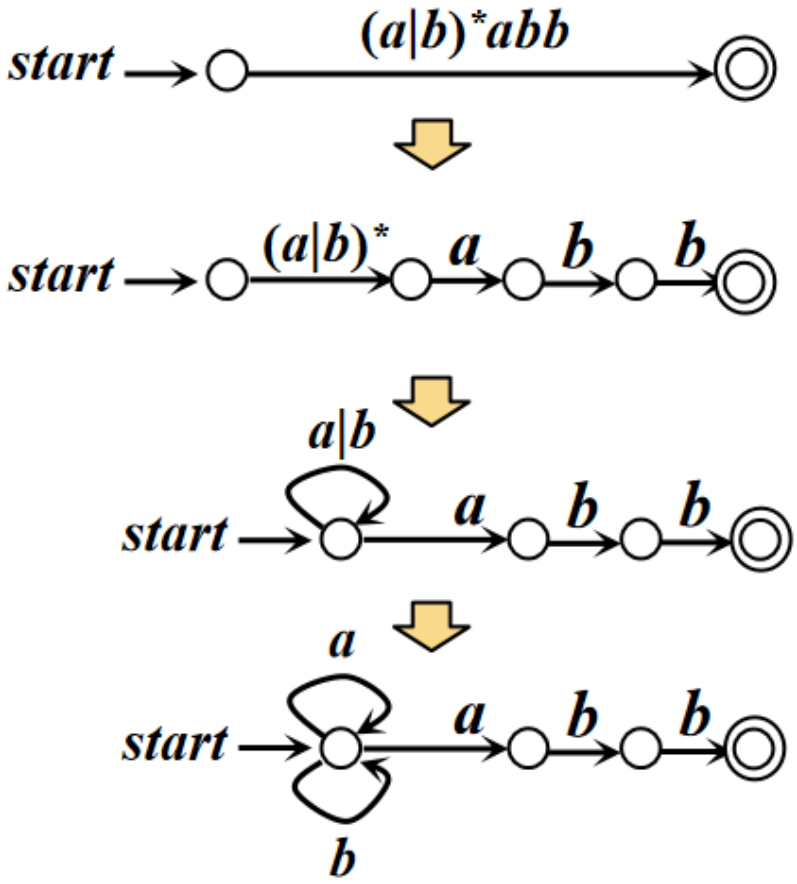
2. 对于 **rs**, 有 ε-NFA:



3. 对于 $\mathbf{r}^*$, 有 ε-NFA:

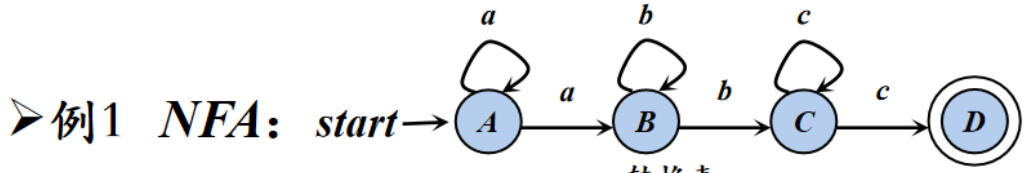


例:$r=(a|b)^*abb$ 对应的NFA



- **NFA到DFA的转换**

➢例1 NFA：



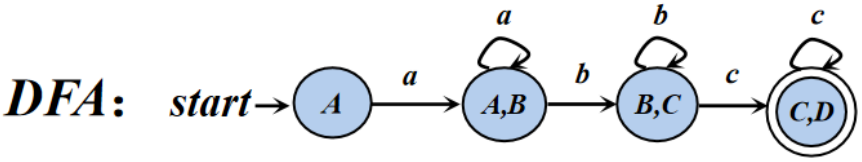
转换表

状态 \ 输入	a	b	c
A	{A,B}	Ø	Ø
B	Ø	{B,C}	Ø
C	Ø	Ø	{C,D}
D	Ø	Ø	Ø

DFA的每个状态都是一个由NFA中的状态构成的集合,即NFA状态集合的一个子集

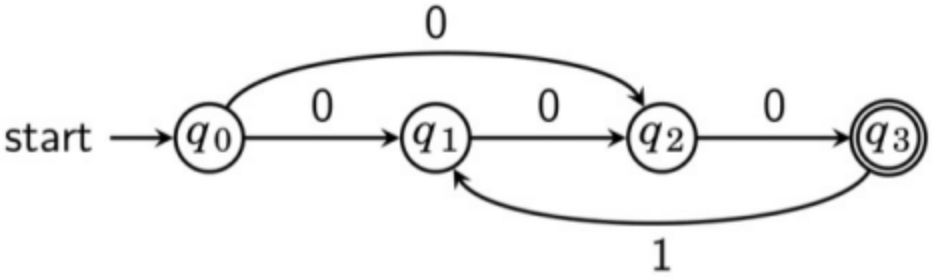
$r = aa^*bb^*cc^*$

DFA：



简答 ch2-4-计算 ★

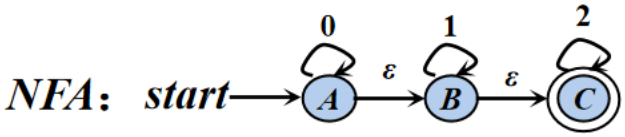
用子集构造法将该NFA转换为等价的DFA。



	0	1
$\rightarrow q_0$	$\{q_1, q_2\}$	$\varnothing$
q_1	$\{q_2\}$	$\varnothing$
q_2	$\{q_3\}$	$\varnothing$
$*q_3$	$\varnothing$	$\{q_1\}$

$\rightarrow\{q_0\}$	$\{q_1,q_2\}$	ϕ
$\{q_1,q_2\}$	$\{q_2,q_3\}$	ϕ
ϕ	ϕ	ϕ
$*\{q_2,q_3\}$	$\{q_3\}$	$\{q_1\}$
$*\{q_3\}$	ϕ	$\{q_1\}$
$\{q_1\}$	$\{q_2\}$	ϕ
$\{q_2\}$	$\{q_3\}$	ϕ

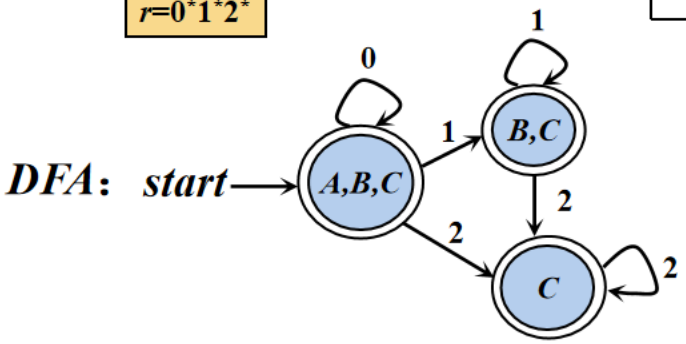
例2：从带有ε-边的NFA到DFA的转换



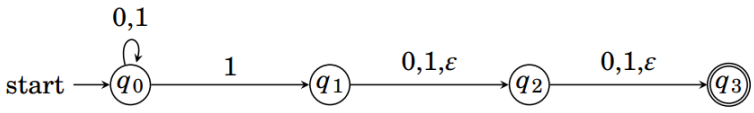
转换表

状态 \ 输入	0	1	2
A	{A,B,C}	{B,C}	{C}
B	Ø	{B,C}	{C}
C	Ø	Ø	{C}

$r=0^*1^*2^*$

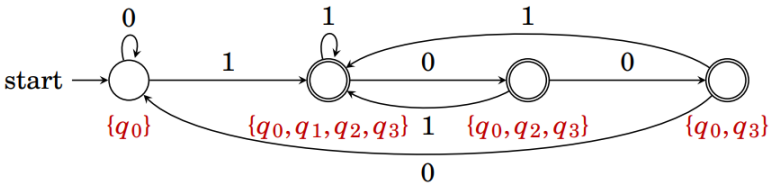


续例 12. 将下图 L 的 ε-NFA, 转为等价的 DFA.



	0	1	ε	ECLOSE(␣)
$\to q_0$	$\{q_0\}$	$\{q_0,q_1\}$	$\varnothing$	$\{q_0\}$
q_1	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$	$\{q_1,q_2,q_3\}$
q_2	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_2,q_3\}$
$*q_3$	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$

当前状态闭包



	0	1
$\to\{q_0\}$	$\{q_0\}$	$\{q_0,q_1,q_2,q_3\}$
$*\{q_0,q_1,q_2,q_3\}$	$\{q_0,q_2,q_3\}$	$\{q_0,q_1,q_2,q_3\}$
$*\{q_0,q_2,q_3\}$	$\{q_0,q_3\}$	$\{q_0,q_1,q_2,q_3\}$
$*\{q_0,q_3\}$	$\{q_0\}$	$\{q_0,q_1,q_2,q_3\}$

增加 q_0,q_1 以及加进来每一个的闭包

子集构造法（*subset construction*）

- 输入：*NFA N*
- 输出：接收同样语言的*DFA D*
- 方法：一开始，*ε-closure*（s_0） 是*Dstates* 中的唯一状态，且它未加标记；

```
while（在Dstates中有一个未标记状态T）  {
        给T加上标记；
        for（每个输入符号a）  {
            U = ε-closure(move(T, a));
            if ( U不在Dstates中)
                将U加入到Dstates中，且不加标记；
            Dtran[T, a]=U ;
        }
}
```

操作	描述
ε-closure (s)	能够从*NFA*的状态*s*开始只通过*ε*转换到达的*NFA*状态集合
ε-closure (T)	能够从*T* 中的某个*NFA*状态 *s*开始只通过*ε*转换到达的*NFA*状态集合，即$U_{s \in T}$ *ε-closure (s)*
move(T , a)	能够从*T* 中的某个状态 *s*出发通过标号为*a*的转换到达的*NFA*状态的集合

计算 *ε-closure (T)*

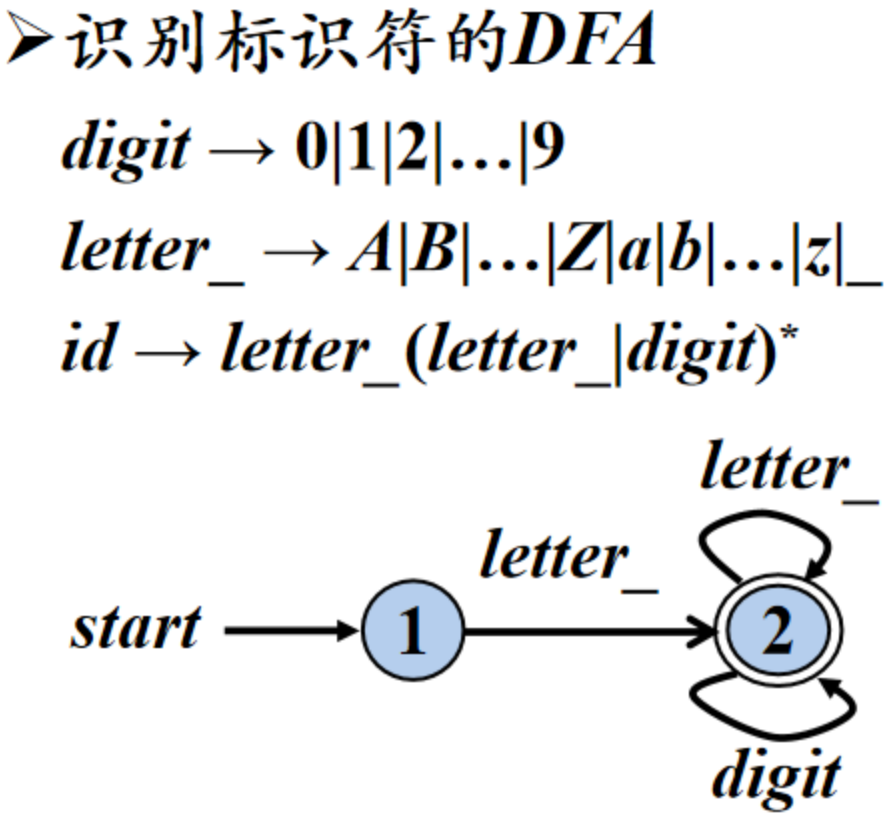
```
将T的所有状态压入stack中；
将ε-closure （T）  初始化为 T ；
while（stack非空） {
        将栈顶元素 t 给弹出栈中；
        for（每个满足如下条件的u ： 从t出发有一个标号为ε的转换到达状态u)
            if ( u不在ε-closure （T）中） {
                将u加入到ε-closure （T）中；
                将u压入栈中；
            }
}
```

3.2.4 识别单词的DFA

1. 以0开头。
2. 可以包含字符0、1和2。
3. 不包含连续的1。

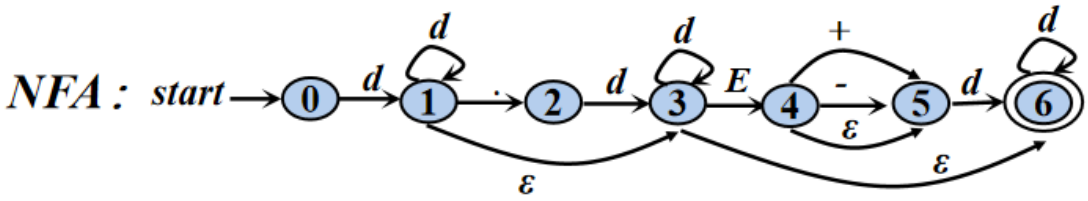
满足这些条件的正则表达式如下：

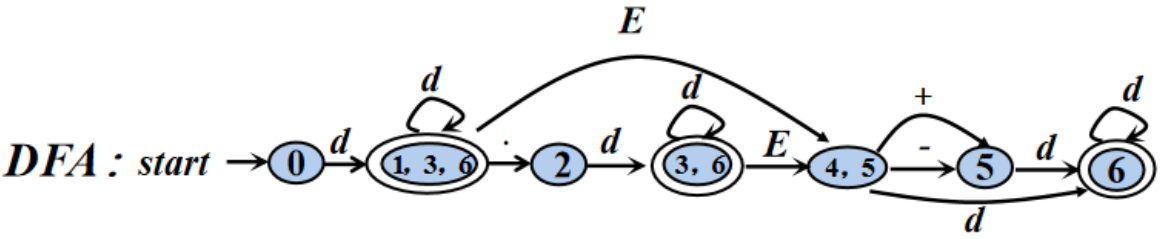
```regex
0(0|2|10|12)*
```



识别无符号数的DFA

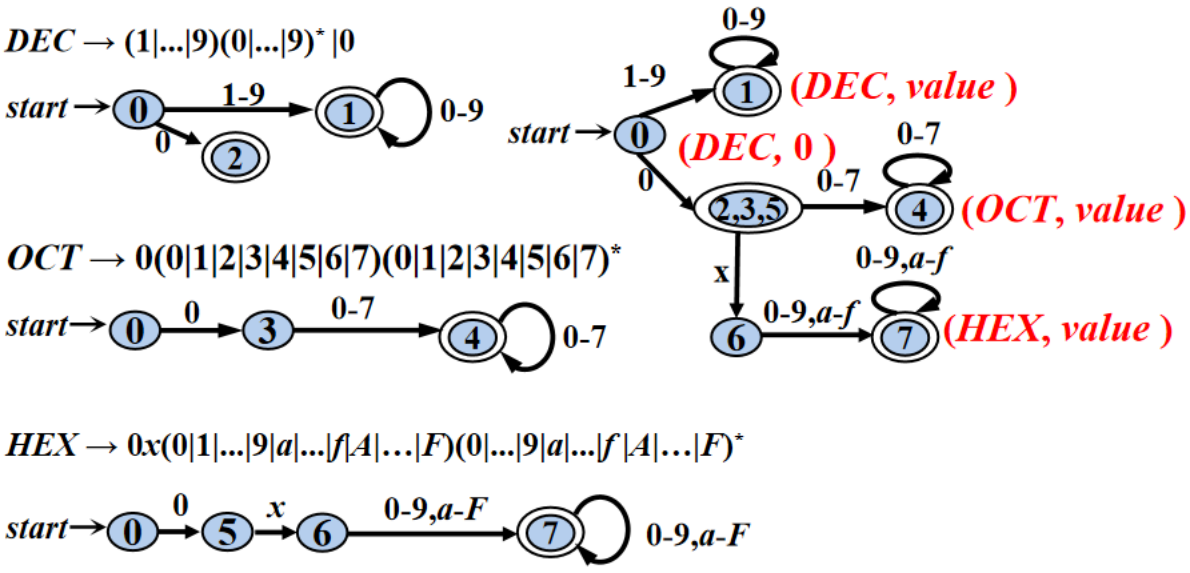
- $digit \rightarrow 0|1|2|\ldots|9$
- $digits \rightarrow digit\ digit^*$
- $optionalFraction \rightarrow .digits|\varepsilon$
- $optionalExponent \rightarrow (\ E(+|-|\varepsilon)digits\ )|\varepsilon$
- $number \rightarrow digits\ \ optionalFraction\ \ optionalExponent$



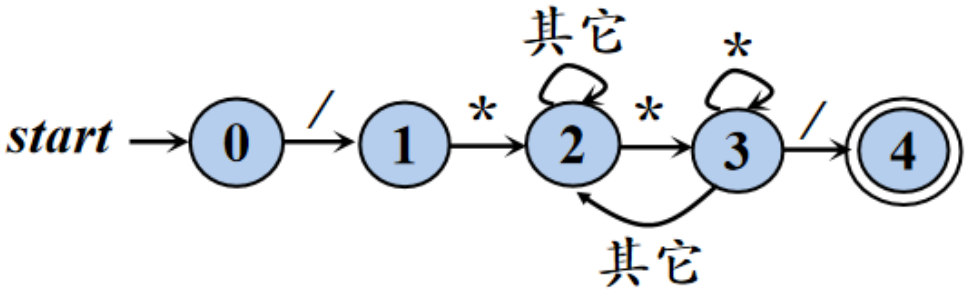


1. **digit** → 0|1|2|...|9
    - 表示一个数字是0到9中的任意一个。
2. **digits** → digit digit*
    - 表示一个或多个数字。
3. **optionalFraction** → .digits|ε
    - 表示可选的小数部分，可以是小数点后跟随一个或多个数字，或者没有小数部分（ε）。
4. **optionalExponent** → (E(+|-|ε)digits)|ε
    - 表示可选的指数部分，可以是字符E后跟随一个可选的正负号，再跟随一个或多个数字，或者没有指数部分（ε）。
5. **number** → digits optionalFraction optionalExponent
    - 表示一个数，可以有整数部分、小数部分和指数部分。

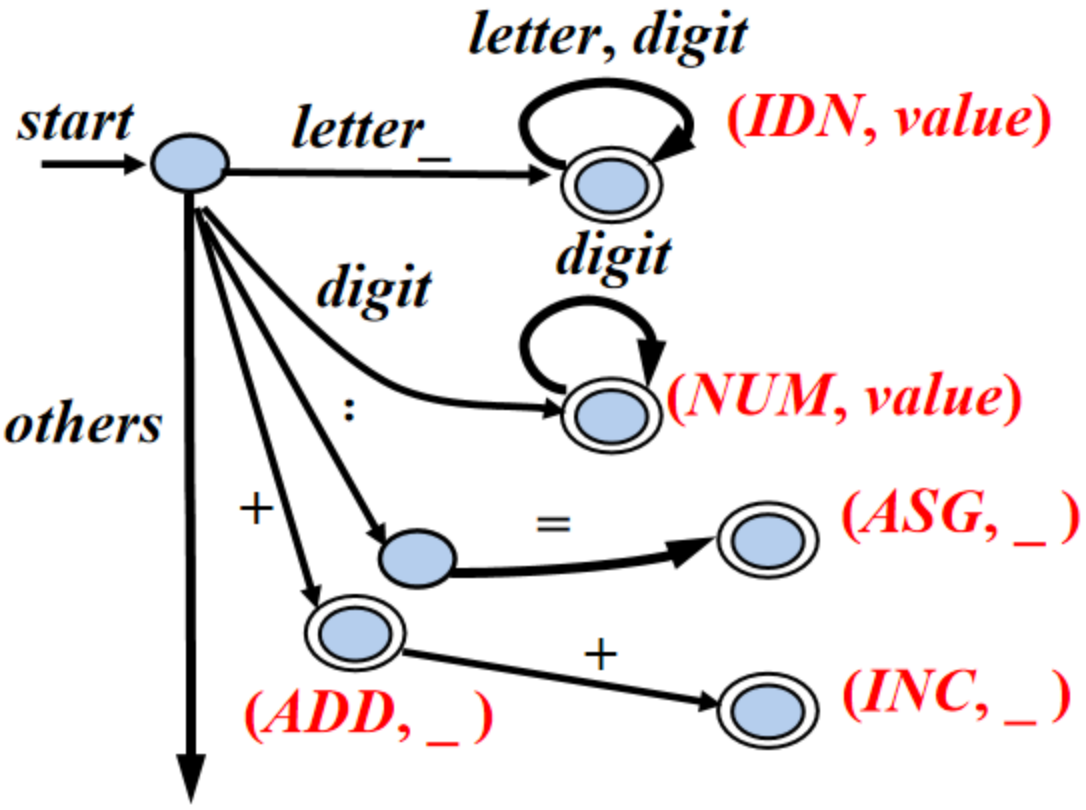
识别各进制无符号整数的*DFA*



识别注释的*DFA*



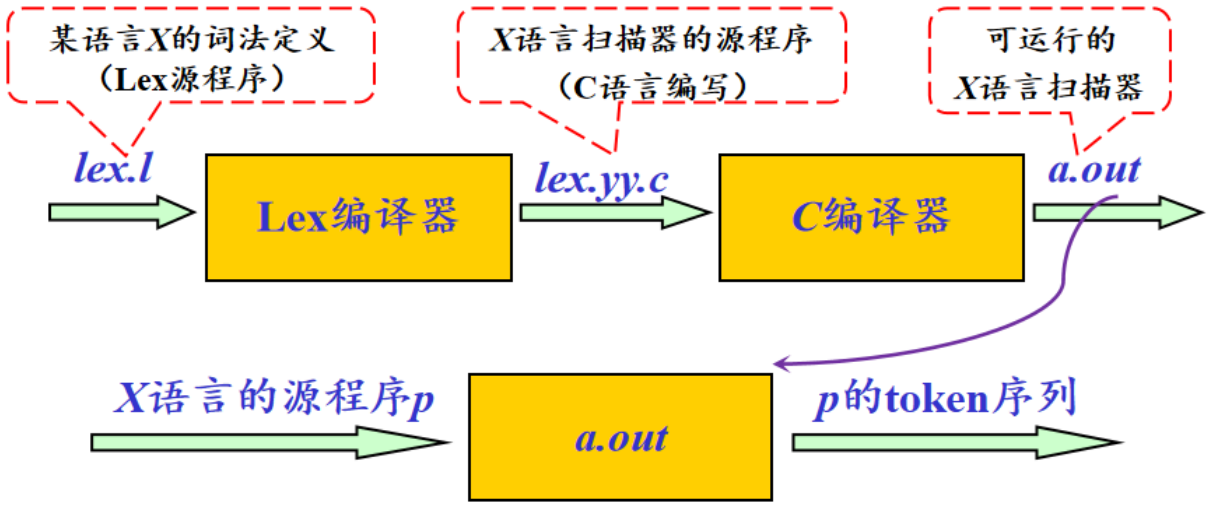
识别*Token*的*DFA*



3.4 词法分析阶段的错误处理

- **词法错误的类型**
  - 单词拼写错误
  - 非法字符
- **词法错误检测**
  - 如果当前状态与当前输入符号在转换表对应项中的信息为空，而当前状态又**不是终止状态**，则调用**错误处理程序**
    1. 非终止状态
    2. 在该状态下后续输入没有匹配的状态转化
- **错误处理**
  - 查找已扫描字符串中**最后一个对应于某终态**的字符
    - 如果**找到了**，将该字符与其前面的字符识别成一个单词。然后将输入指针**退回**到该字符，扫描器重新**回到初始状态**，继续识别下一个单词
    - 如果**没找到**，则确定出错，采用**错误恢复策略**
- **错误恢复策略**
  - 最简单的错误恢复策略：“恐慌模式 (panic mode)”恢复
    - 从剩余的输入中**不断删除字符**，直到词法分析器能够在剩余输入的开头发现一个正确的字符为止

3.4 词法分析器生成工具Lex



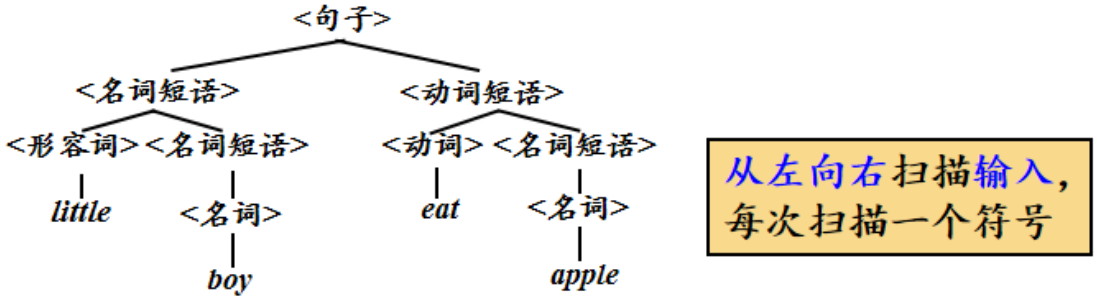
扫描器自动生成的意义

- Lex的构成加快了分析器的实现速度
  - 程序员只需在很高的模式层次上描述软件，就可以依赖自动生成工具来生成详细的代码
- 修改扫描器的工作变得更加简单
  - 只需修改那些受到影响的模式，无需改写整个程序

第四章 语法分析

- 语法分析的主要任务
  - 根据给定的**文法**，**识别输入句子的各个成分**，从而构造出句子的**分析树**

- 大部分程序设计语言的语法构造可以用CFG（上下文无关语法）来描述，CFG**以token作为终结符**
- 大部分语法分析器都期望文法是**无二义性**的，否则，就不能为一个句子构造**唯一**的语法分析树



- 自顶向下的分析(*Top-Down Parsing*)
  - 从分析树的顶部（根节点）向底部（叶节点）构造分析树
  - 从文法开始符号S推导出串w
- 自底向上的分析(*Bottom-up Parsing*)
  - 从分析树的底部（叶节点）向顶部（根节点）构造分析树
  - 将一个串w归约为文法开始符号S

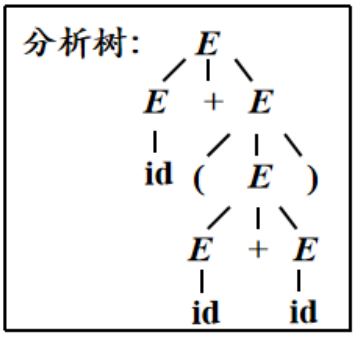
- 最高效的自顶向下和自底向上方法只能处理某些文法子类，但是其中的某些子类，特别是*LL*和*LR*文法，其表达能力足以描述现代程序设计语言的大部分语法构造

4.1 自顶向下的分析

- 从分析树的顶部（根节点）向底部（叶节点）方向构造分析树
- 可以看成是从文法开始符号S推导出词串w的过程
  - 例

文法
① $E \to E+E$
② $E \to E*E$
③ $E \to (E)$
④ $E \to \mathrm{id}$

输入
$\mathrm{id}+(\mathrm{id}+\mathrm{id})$



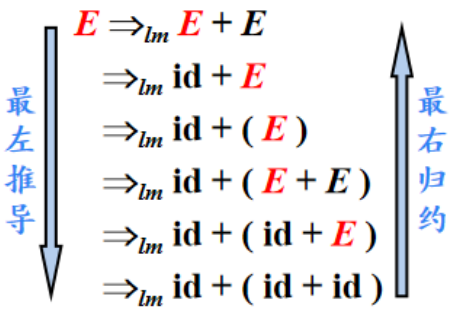
推导：$E \Rightarrow E+E \Rightarrow \mathrm{id}+E \Rightarrow \mathrm{id}+(E) \Rightarrow \mathrm{id}+(E+E) \Rightarrow \mathrm{id}+(\mathrm{id}+E) \Rightarrow \mathrm{id}+(\mathrm{id}+\mathrm{id})$

- 推导的每一步，都需要做两个选择
  - 替换当前句型中的哪个非终结符
  - 用该非终结符的哪个候选式进行替换

- 在最左推导中，总是选择每个句型的最左非终结符进行替换
  - 例

文法
① $E \to E+E$
② $E \to E*E$
③ $E \to (E)$
④ $E \to \mathrm{id}$

输入
$\mathrm{id}+(\mathrm{id}+\mathrm{id})$



  - 如果$S \Rightarrow_{lm}^{*} \alpha$，则称$\alpha$是当前文法的最左句型(*leftmost-sentential form*)

- 在最右推导中，总是选择每个句型的最右非终结符进行替换
  - 例

文法
① $E \to E + E$
② $E \to E * E$
③ $E \to (E)$
④ $E \to id$

输入
id + (id + id)

$$E \Rightarrow_{rm} E + E$$
$$\Rightarrow_{rm} E + (E)$$
$$\Rightarrow_{rm} E + (E + E)$$
$$\Rightarrow_{rm} E + (E + id)$$
$$\Rightarrow_{rm} E + (id + id)$$
$$\Rightarrow_{rm} id + (id + id)$$

最右推导 ↓　　↑ 最左归约

最右句型也称为规范句型

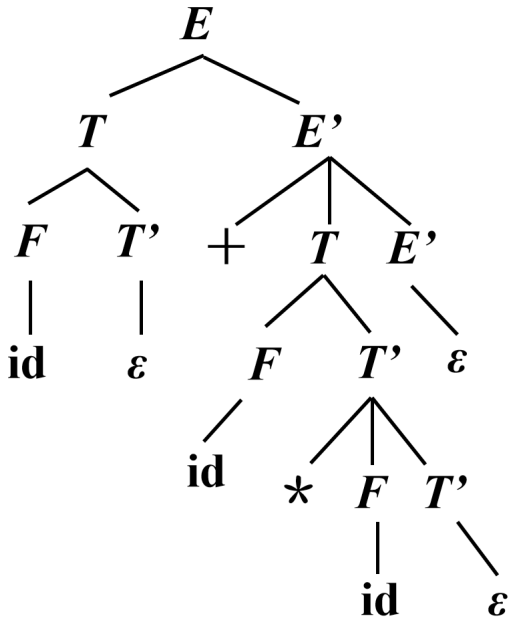
  - 如果$S \Rightarrow^*_{rm} \alpha$，则称$\alpha$是当前文法的最右句型(*rightmost-sentential form*)
  - 在自底向上的分析中，总是采用最左归约的方式，因此把最左归约称为规范归约，而最右推导相应地称为规范推导

- **最左推导和最右推导具有唯一性**
- **自顶向下的语法分析采用最左推导方式**

- 总是选择每个句型的最左非终结符进行替换
- 根据输入流中的当前终结符，选择最左非终结符的一个候选式

- 文法
  - ① $E \rightarrow T\,E'$
  - ② $E' \rightarrow + T\,E' \mid \varepsilon$
  - ③ $T \rightarrow F\,T'$
  - ④ $T' \rightarrow * F\,T' \mid \varepsilon$
  - ⑤ $F \rightarrow (\,E\,) \mid id$
- 输入

  id + id * id



1. 自顶向下语法分析的通用形式（最左推导）

- 递归下降分析 (*Recursive-Descent Parsing*)
  - 由一组过程组成，每个过程对应一个非终结符
  - 从文法开始符号S对应的过程开始，其中递归调用文法中其它非终结符对应的过程。如果S对应的过程体恰好扫描了整个输入串，则成功完成语法分析

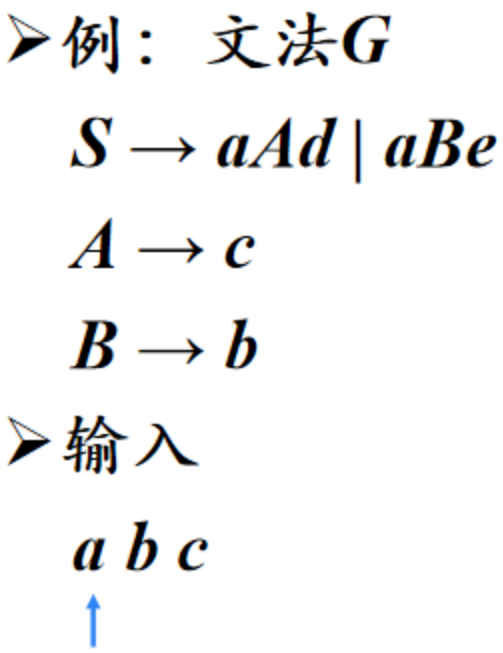
```
    void A( ) {
1)    选择一个A产生式，A →X1 X2 … Xk ；
2)    for ( i = 1 to k ) {
3)        if ( Xi是一个非终结符号)
4)            调用过程 Xi( ) ;
5)        else if ( Xi 等于当前的输入符号a)
6)            读入下一个输入符号;
7)        else /* 发生了一个错误 */ ;
        }
    }
```

可能需要回溯(*backtracking*)，导致效率较低

需要回溯的分析（不确定的分析）

2. 自顶向下分析存在的问题

- **问题一**：**同一非终结符**的多个候选式存在**共同前缀**，将导致**回溯**现象



- **问题二**：左递归文法会使递归下降分析器陷入**无限循环**

➢ 文法G

$E \to E+T \mid E-T \mid T$

$T \to T*F \mid T/F \mid F$

$F \to (E) \mid \mathrm{id}$

$$E \Rightarrow E+T \Rightarrow E+T+T \Rightarrow E+T+T+T \Rightarrow \dots$$

➢ 输入

id + id * id

↑

如果一个文法中有一个非终结符A使得对某个串α存在一个推导$A \Rightarrow^{+} A\alpha$，那么这个文法就是左递归的
含有$A \to A\alpha$形式产生式的文法称为是直接左递归的 (*immediate left recursive*)
经过两步或两步以上推导产生的左递归称为是间接左递归的

3. 消除直接左递归（解决问题二）

$$A \to A\alpha \mid \beta (\alpha \neq \varepsilon, \beta \text{不以} A \text{开头})$$

$r=\beta\alpha^{*}$

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \Rightarrow \dots \Rightarrow A\alpha\dots\alpha \Rightarrow \beta\alpha\dots\alpha$$

$$A \to \beta A'$$

$$A' \to \alpha A' \mid \varepsilon$$

事实上，这种消除过程就是把左递归转换成了右递归

➢ 例

$E \to E+T \mid T$ （α = $+T$，β = T）

$T \to T*F \mid F$ （α = $*F$，β = F）

$F \to (E) \mid \mathrm{id}$

$$E \to TE'$$

$$E' \to +TE' \mid \varepsilon$$

$$T \to FT'$$

$$T' \to *FT' \mid \varepsilon$$

$$F \to (E) \mid \mathrm{id}$$

$$A' \Rightarrow \alpha A' \Rightarrow \alpha\alpha A' \Rightarrow \alpha\alpha\alpha A' \Rightarrow \dots \Rightarrow \alpha\dots\alpha A' \Rightarrow \alpha\dots\alpha$$

消除直接左递归的一般形式

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

$$(\alpha_i \neq \varepsilon, \beta_j \text{不以} A \text{开头})$$

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

消除左递归是要付出代价的——引进了一些非终结符和ε_产生式

4. 消除间接左递归（解决问题二）

➢ 例 $S \to Aa \mid b$

$A \to Ac \mid Sd \mid \varepsilon$

$S \Rightarrow Aa$
$\Rightarrow Sda$

➢ 将S的定义代入A-产生式，得：

$A \to Ac \mid Aad \mid bd \mid \varepsilon$

转换成直接左递归

➢ 消除A-产生式的直接左递归，得：

$A \to bdA' \mid A'$

$A' \to cA' \mid adA' \mid \varepsilon$

消除间接左递归算法

- 输入：不含循环推导（即形如$A \Rightarrow^{+} A$的推导）和ε-产生式的文法G
- 输出：等价的无左递归文法
- 方法：

1) 按照某个顺序将非终结符号排序为A_1，A_2，…，A_n .
2) for (从1到n的每个i) {
3) for (从1到i -1的每个j) {
4) 将每个形如$A_i \rightarrow A_j \gamma$的产生式替换为产生式组$A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$，其中$A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$，是所有的$A_j$产生式
5) }
6) 消除A_i产生式之间的直接左递归
7) }

注：注意顺序，只有在当前表达式之前的表达式可以用于替换

5. 提取左公因子（解决问题一）

- 例
  - 文法G
    - $S \rightarrow aAd \mid aBe$
    - $A \rightarrow c$
    - $B \rightarrow b$

通过改写产生式来推迟决定，等读入了足够多的输入，获得足够信息后再做出正确的选择

  - 文法G'
    - $S \rightarrow aS'$
    - $S' \rightarrow Ad \mid Be$
    - $A \rightarrow c$
    - $B \rightarrow b$

提取左公因子算法

- 输入：文法G
- 输出：等价的提取了左公因子的文法
- 方法：

对于每个非终结符A，找出它的两个或多个选项之间的最长公共前缀α。如果$\alpha \neq \varepsilon$，即存在一个非平凡的(*nontrivial*)公共前缀，那么将所有A-产生式

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m$$

替换为

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

其中，γ_i表示所有不以α开头的产生式体；A'是一个新的非终结符。不断应用这个转换，直到每个非终结符的任意两个产生式体都没有公共前缀为止

习题

- T 4.2 (1)

➢为下面的每一个文法设计一个预测分析器，并给出预测分析表。你可能先要对文法进行提取左公因子或消除左递归的操作。计算各文法的FIRST和FOLLOW集合。

➢(1) S → 0 S 1 | 0 1

提取左公因子
S→0A
A→S1|1

	FIRST	FOLLOW
S	{0}	{#, 1}
A	{0, 1}	{#, 1}

- T 4.3

➢为下面的文法设计一个预测分析器，并给出预测分析表。你可能先要对文法进行提取左公因子或消除左递归的操作。计算文法的FIRST和FOLLOW集合。

S → S S + | S S * | a

提取左公因子
S→ SSA|a
A→ +|*

消除左递归
S→ aS'
S'→ SAS'|ε
A→ +|*

A→Aα|β
A→βA'
A'→αA'|ε

	FIRST	FOLLOW
S	{a}	{#, +, *}
S'	{a, ε}	{#, +, *}
A	{+, *}	{a, #, +, *}

4.2 预测分析法

- 预测分析是递归下降分析技术的一个特例，通过在输入中向前看固定个数（通常是一个）符号来选择正确的A-产生式。
  - 可以对某些文法构造出向前看k个输入符号的预测分析器，该类文法有时也称为$LL(k)$ 文法类
- 预测分析不需要回溯，是一种确定的自顶向下分析方法

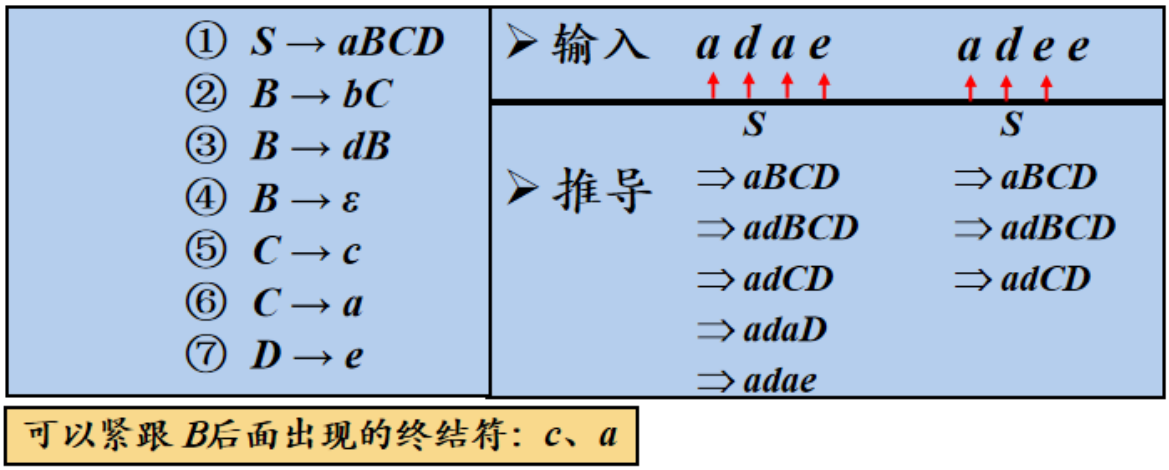
4.2.1 LL(1) 文法

- 预测分析法的工作过程
  - 从文法开始符号出发，在每一步推导过程中根据当前句型的最左非终结符A和当前输入符号a，选择正确的A-产生式。为保证分析的确定性，选出的候选式必须是唯一的。
- S_文法（简单的确定性文法，*Korenjak & Hopcroft*，1966）

每个产生式的右部都以终结符开始

同一非终结符的各个候选式的首终结符都不同

S_文法不含ε产生式



➢ 什么时候使用ε产生式?

➢ 如果当前某非终结符A与当前输入符a不匹配时，若存在$A\to\varepsilon$，可以通过检查a是否可以出现在A的后面，来决定是否使用产生式$A\to\varepsilon$（若文法中无$A\to\varepsilon$，则应报错）

串首终结符集（FIRST）

- 串首终结符
  - 第一个符号
  - 终结符
- 给定一个文法符号串α， α的串首终结符集**FIRST(α)**被定义为可以从α推导出的**所有串首终结符**构成的集合。如果α →* ε，那么 **ε 也在FIRST(α)**中

➢ 对于 $\forall\alpha\in(V_T\cup V_N)^+$, $FIRST(\alpha)=\{a\mid \alpha\Rightarrow^* a\beta,\ a\in V_T,\ \beta\in(V_T\cup V_N)^*\}$;

➢ 如果 $\alpha\Rightarrow^*\varepsilon$，那么 $\varepsilon\in FIRST(\alpha)$

FIRST(α)的计算

$\alpha = X_1X_2\cdots X_k$

➢ $FIRST(\alpha)$：首终结符集。α能够推出的所有终结符串中位于串首的那些终结符构成的集合

$$= FIRST(X_1) \cup FIRST(X_2) \cup FIRST(X_3) \cup \cdots$$
$$\text{if } X_1 \Rightarrow^* \varepsilon \qquad \text{if } X_2 \Rightarrow^* \varepsilon$$

$\alpha \Rightarrow^* \varepsilon \quad \Leftrightarrow \quad \varepsilon \in FIRST(\alpha)$

$$FIRST(X_i) = \begin{cases} \{X_i\} & \text{if } X_i \in V_T \quad \text{终结符} \\ \text{通过} X_i\text{-产生式右部求} & \text{if } X_i \in V_N \quad \text{非终结符} \end{cases}$$

➢ 例

① $E \to TE'$ $\quad FIRST(E) = \{ (, id \}$

② $E' \to +TE' \mid \varepsilon$ $\quad FIRST(E') = \{ +, \varepsilon \}$

③ $T \to FT'$ $\quad FIRST(T) = \{ (, id \}$

④ $T' \to *FT' \mid \varepsilon$ $\quad FIRST(T') = \{ *, \varepsilon \}$

⑤ $F \to (E) \mid id$ $\quad FIRST(F) = \{ (, id \}$

注：要注意如果产生式右部以非终结符开头，需要看该终结符是否能推出 ε

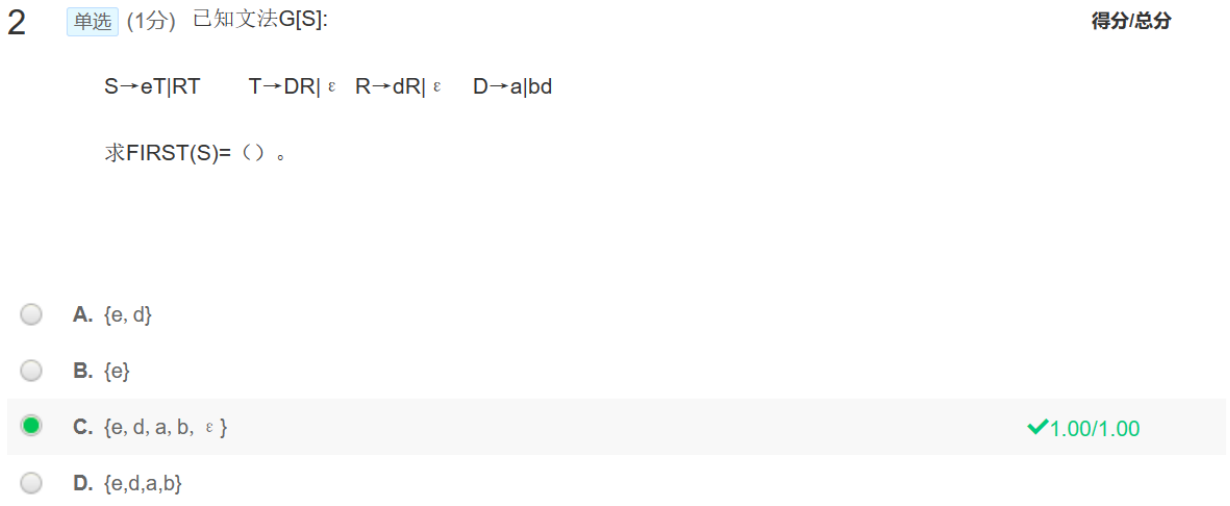
计算文法符号FIRST(X)的算法

- 不断应用下列规则，直到没有新的终结符或ε可以被加入到任何FIRST集合中为止
  - 如果X是一个终结符，那么FIRST (X) = { X }
  - 如果X是一个非终结符，且 X→Y1...Yk∈P (k≥1)
    1. 若FIRST（Y1）中不含ε，将FIRST（Y1）加入FIRST（X）中
    2. 若FIRST（Y1）中含有ε，将FIRST（Y1）∪ FIRST（Y2）-{ε}加入FIRST（X）中
    3. 对后续Yi重复上述两个步骤，直到Yk
    4. 若所有Yi都包含ε，将ε加入到FIRST(X)中
  - 如果 **X→ε**∈P，那么将ε加入到FIRST(X)中

计算串X1X2 ...Xn的FIRST 集合

1. 向FIRST(X1X2 ...Xn)加入FIRST(X1)中所有的**非ε符号**
2. 如果ε在FIRST(X1)中，再加入FIRST(X2)中的所有非ε符号；
3. 如果ε在FIRST(X1)和FIRST(X2)中，再加入FIRST(X3)中的所有非ε符号，以此类推
4. 最后，如果**对所有的i，ε都在FIRST(Xi)中**，那么**将ε加入到FIRST(X1X2 ...Xn) 中**

习题



> 💡 **注：**
>
> - **同时存在 `S→RT` 和 `R→ε` 这时候就要考虑T的FIRST集了**
> - **要注意只有当S能推出 `ε` 的时候才将 `ε` 加入到S的FIRST集中，也就是说在本题中需要同时存在 `R→ε` 和 `T→ε` 才可以**

非终结符的后继符号集（FOLLOW）

➢ 可能在某个句型中紧跟在A后边的终结符a的集合，记为*FOLLOW(A)*

$FOLLOW(A)=\{a \mid S \Rightarrow^{*} \alpha A a \beta, a \in V_T, \ \alpha,\beta \in (V_T \cup V_N)^{*}\}$

例

(1) $S \rightarrow aBCD$

(2) $B \rightarrow bC$ ← 输入 b

(3) $B \rightarrow dB$ ← d

(4) $B \rightarrow \varepsilon$ ← $\{a, c\}$

(5) $C \rightarrow c$

(6) $C \rightarrow a$

(7) $D \rightarrow e$

$FOLLOW(B) = \{a,c\}$

如果*A*是某个句型的的最右符号，则将结束符“$”添加到*FOLLOW(A)*中

计算非终结符A的FOLLOW(A)

例

① $E \rightarrow TE'$ $FIRST(E) = \{ (, id \}$ $FOLLOW(E) = \{ \$,) \}$

② $E' \rightarrow +TE' \mid \varepsilon$ $FIRST(E') = \{ +, \varepsilon \}$ $FOLLOW(E') = \{ \$,) \}$

③ $T \rightarrow FT'$ $FIRST(T) = \{ (, id \}$ $FOLLOW(T) = \{ +, \$,) \}$

④ $T' \rightarrow *FT' \mid \varepsilon$ $FIRST(T') = \{ *, \varepsilon \}$ $FOLLOW(T') = \{ +, \$,) \}$

⑤ $F \rightarrow (E) \mid id$ $FIRST(F) = \{ (, id \}$ $FOLLOW(F) = \{ *, +, \$,) \}$

- 一个产生式一个产生式看

- 不断应用下列规则，直到没有新的终结符可以被加入到任何*FOLLOW*集合中为止
  - 将$放入*FOLLOW*(*S*)中，其中*S*是开始符号，$是输入右端的结束标记
  - 如果存在一个产生式*A*→*αBβ*，那么*FIRST*(*β*)中除ε之外的所有符号都在*FOLLOW*(*B*)中
  - 如果存在一个产生式*A*→*αB*，或存在产生式*A*→*αBβ*且*FIRST*(*β*)包含ε，那么*FOLLOW*(*A*)中的所有符号都在*FOLLOW*(*B*)中

3 单选 (1分) 已知文法G[S]:

S→eT|RT T→DR|ε R→dR|ε D→a|bd

求FOLLOW(D)=（）。

d, $

FOLLOW(D) ← FOLLOW(T) = $

A. {a,d}

B. {d,$}

C. {d,e}

D. {d, ε }

产生式的的可选集（SELECT）

- 产生式A→β的可选集是指**可以选用该产生式进行推导时对应的输入符号的集合**，记为**SELECT(A→β)**
- 当输入符号在某个产生式可选集中，就可以用这个产生式

- SELECT(A→aβ) = { a }
- **SELECT(A→ε)=FOLLOW(A)**

➢ q_文法

➢ 每个产生式的右部或为ε，或以终结符开始

➢ 具有相同左部的产生式有不相交的可选集

q_文法不含右部以非终结符打头的产生式

产生式A→α的可选集

➢ 产生式$A \to \alpha$的可选集SELECT

➢ 如果$\varepsilon \notin FIRST(\alpha)$，那么$SELECT(A \to \alpha) = FIRST(\alpha)$

➢ 如果$\varepsilon \in FIRST(\alpha)$，那么$SELECT(A \to \alpha) = (FIRST(\alpha) - \{\varepsilon\}) \cup FOLLOW(A)$

X	FIRST(X)	FOLLOW(X)
E	(id	$)
E'	+ ε	$)
T	(id	+) $
T'	* ε	+) $
F	(id	* +) $

表达式文法是 LL(1) 文法

(1) $E \to T\,E'$　　SELECT (1)= { (id }

(2) $E' \to + T\,E'$　　SELECT (2)= { + }

(3) $E' \to \varepsilon$　　SELECT (3)= { $) }

(4) $T \to F\,T'$　　SELECT (4)= { (id }

(5) $T' \to * F\,T'$　　SELECT (5)= { * }

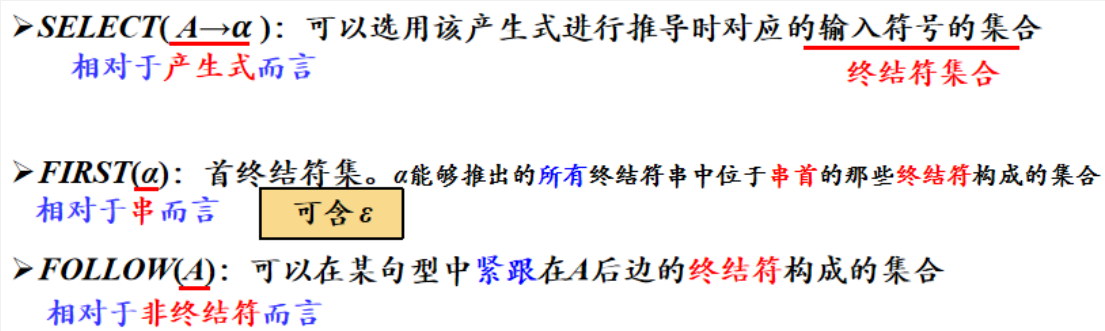
(6) $T' \to \varepsilon$　　SELECT (6)= { +) $ }

(7) $F \to (\,E\,)$　　SELECT (7)= { (}

(8) $F \to$ id　　SELECT (8)= { id }

总结

- **S_文法**（每个产生式的右部都以**终结符**开始，同一非终结符的各个候选式的**首终结符**都不同，S文法不含**ε产生式**）
- **q_文法** （每个产生式的右部或为**ε** ，或以**终结符**开始 → 具有相同左部的产生式有**不相交**的可选集 。）
- FOLLOW ：非终结符A的（在某个句型中紧跟在A后边的终结符a的集合）
- SELECT ：产生式 A→α（产生式A→β的可选集是指可以选用该产生式进行推导时对应的输入符号的集合，SELECT(A→aβ) = { a }，**SELECT(A→ε)=FOLLOW(A)**）
- FIRST ：串 α→X1X2X3X4....（给定一个**文法符号串α**， α的**串首终结符集FIRST(α)被定义为可以从α推导出的所有串首终结符构成的集合**。如果α ⇒ ε， 那么ε也在FIRST(α)中。如果X是一个终结符，那么FIRST(X) = {X}，非终结符就递归着看，如果 X→ε∈P，那么将ε加入到FIRST(X) 中）
- 先求FIRST，再求FOLLOW，最后求SELECT
- **三个集合里面都是终结符**
- **只有FIRST集中会有空串！！！**



LL（1）文法

- **一个文法G，若G的LL（1）分析表中不含多重条目，则称他是LL（1）文法**

- 文法G是$LL(1)$的，当且仅当G的任意两个具有相同左部的产生式$A \to \alpha \mid \beta$满足下面的条件：
  - 不存在终结符a使得α和β都能够推导出以a开头的串
  - α和β至多有一个能推导出ε
  - 如果$\beta \Rightarrow^* \varepsilon$，则$FIRST(\alpha) \cap FOLLOW(A) = \Phi$；
    如果$\alpha \Rightarrow^* \varepsilon$，则$FIRST(\beta) \cap FOLLOW(A) = \Phi$；

同一非终结符的各个产生式的可选集互不相交

可以为$LL(1)$文法构造预测分析器

- **如果β→*ε，SELECT（A→α）=FIRST（α）**

- 第一个“L”表示从左（Left）向右扫描输入
- 第二个“L”表示产生最左（Left）推导
- “1”表示在每一步中只需要向前看一个输入符号来决定语法分析动作

预测分析表

	产生式	SELECT
E	$E\to TE'$	(id
E'	$E'\to +TE'$	+
	$E'\to\varepsilon$	$)
T	$T\to FT'$	(id
T'	$T'\to *FT'$	*
F	$F\to (E)$	(
	$F\to id$	id

非终结符	输入符号					
	id	+	*	(	)	$
E	$E\to TE'$			$E\to TE'$		
E'		$E'\to +TE'$			$E'\to\varepsilon$	$E'\to\varepsilon$
T	$T\to FT'$			$T\to FT'$		
T'		$T'\to\varepsilon$	$T'\to *FT'$		$T'\to\varepsilon$	$T'\to\varepsilon$
F	$F\to id$			$F\to (E)$		

如何实现预测分析

- **递归的方式**：基于**预测分析表**对**递归下降分析法**进行扩展
- **非递归的方式**：显式地维护一个**栈结构**来模拟**最左推导**过程

习题

5 单选 (1分) 一个文法G，若()，则称它是LL（1）文法。 得分/总分

A. G中产生式不含左公因子

B. G无二义性

C. G中不含左递归

D. G的LL（1）分析表中不含多重定义的条目 ✔1.00/1.00

4.2.2 递归的预测分析法

- 隐式的维护一个栈结构

➢ 是对递归下降分析框架的扩展

```
A( Token )
{
    if Token∈SELECT(A →α1)
        code1;
    if Token∈SELECT(A →α2)
        code2;
    …
    if Token∈SELECT(A →αn)
        coden;
}
```

$A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_n$

$code_1$ | $code_2$ | … | $code_n$

$code_i$; 设 $\alpha_i = X_1 X_2 \ldots X_k$

for (j = 1 to k)

if $X_j \in V_T$ (终结符)
- if $X_j == Token$ then GetNext (Token)
- else ($X_j \neq Token$) Error( )

else ($X_j \in V_N$) X_j(Token)

递归下降语法分析器

(1) <PROGRAM> → program <DECLIST> :<TYPE> ; <STLIST> end

(2) <DECLIST> → id <DECLISTN>

(3) <DECLISTN> → , id <DECLISTN>

(4) <DECLISTN> → ε

(5) <STLIST> → s <STLISTN>

(6) <STLISTN> → ; s <STLISTN>

(7) <STLISTN> → ε

(8) <TYPE> → real

(9) <TYPE> → int

```
program DESCENT;
  begin
    GETNEXT(TOKEN);
    PROGRAM(TOKEN);
    if TOKEN≠'$' then ERROR;
  end
```

SELECT(4)={:}
SELECT(7)={end}

(1) <PROGRAM> → program <DECLIST> :<TYPE> ; <STLIST> end

```
procedure PROGRAM(TOKEN);
    begin
      if TOKEN≠'program' then ERROR;
       GETNEXT(TOKEN);

      DECLIST(TOKEN);

      if TOKEN≠':' then ERROR;
      GETNEXT(TOKEN);

      TYPE(TOKEN)

      if TOKEN≠';' then ERROR;
      GETNEXT(TOKEN);

      STLIST(TOKEN);

      if TOKEN≠'end' then ERROR;
      GETNEXT(TOKEN);
    end
```

(3) <DECLISTN> → , id <DECLISTN>

(4) <DECLISTN> → ε

```
procedure DECLISTN(TOKEN);
        begin
            if TOKEN =',' then
              begin
                GETNEXT(TOKEN);

                if TOKEN≠'id'  then ERROR;
                GETNEXT(TOKEN);

                DECLISTN(TOKEN);
              end
            else  if TOKEN≠':'  then ERROR;
        end
```

(8) <TYPE> → real

(9) <TYPE> → int

```
procedure TYPE(TOKEN);
    begin
        if TOKEN≠'real' and TOKEN≠'int'
            then ERROR;
        GETNEXT(TOKEN);
    end
```

💡 **注：不是LL（1）文法，不可以采用递归的预测分析，无法构造分析器**

1）构造文法　　无二义性

2）改造文法：消除二义性、消除左递归、消除回溯

3）求每个变量的*FIRST*集和*FOLLOW*集，从而求得每个候选式的*SELECT*集

确定性

4）检查是不是*LL*(1) 文法。若是，构造预测分析表

5）对于递归的预测分析，根据预测分析表为每一个非终结符编写一个过程；对于非递归的预测分析，实现表驱动的预测分析算法

习题

if （不）匹配终结符之后GETNEXT（TOKEN）

- T 5.1 (3)

➢ 为下列文法构造递归下降语法分析器（参见SPOC讲义“第4章 语法分析 - 上.pdf”第42~48页）

➢ (3) S → 0 S 1 | 0 1

提取左公因式

S→0A

A→S1 | 1

	FIRST	FOLLOW
S	{0}	{$, 1}
A	{0, 1}	{$, 1}

	SELECT
S→0A	{0}
A→S1	{0}
A→1	{1}

```
program DECENT;
begin
  GETNEXT(TOKEN);
  S(TOKEN);
  if TOKEN≠'$' then ERROR
end.
```

```
program S(TOKEN);
begin
  if TOKEN≠'0' then ERROR;
  GETNEXT(TOKEN);
  A(TOKEN);
end
```

```
program A(TOKEN);
begin
  if TOKEN='0' then
  begin
    S(TOKEN);
    if TOKEN≠'1' then ERROR;
    GETNEXT(TOKEN);
  end
  else if TOKEN≠'1' then ERROR;
    GETNEXT(TOKEN);
end
```

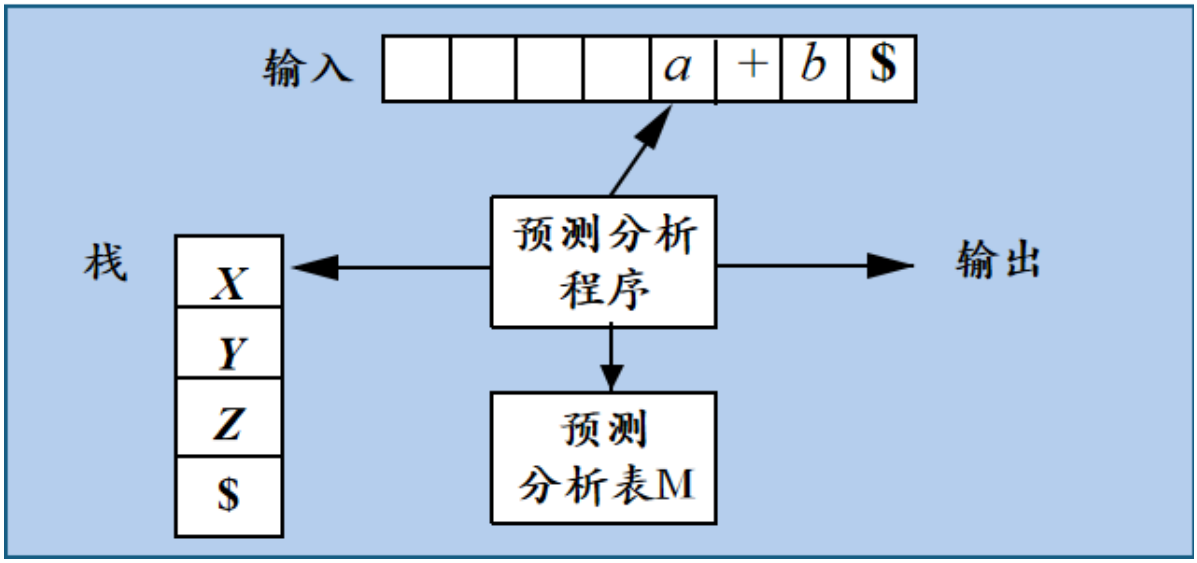
```
program DESCENT;
begin
    GETNEXT(TOKEN);
    S(TOKEN);
    if TOKEN ≠ '$' then ERROR;
end;

procedure S(TOKEN);
begin
    if TOKEN = '0' then
    begin
        GETNEXT(TOKEN);
        A(TOKEN);
    end
    else
        ERROR;
end;

procedure A(TOKEN);
begin
    if TOKEN = '0' then
    begin
        S(TOKEN);
        if TOKEN ≠ '1' then ERROR;
        GETNEXT(TOKEN);
    end
    else if TOKEN = '1' then
    begin
        GETNEXT(TOKEN);
    end
    else
        ERROR;
end;
```

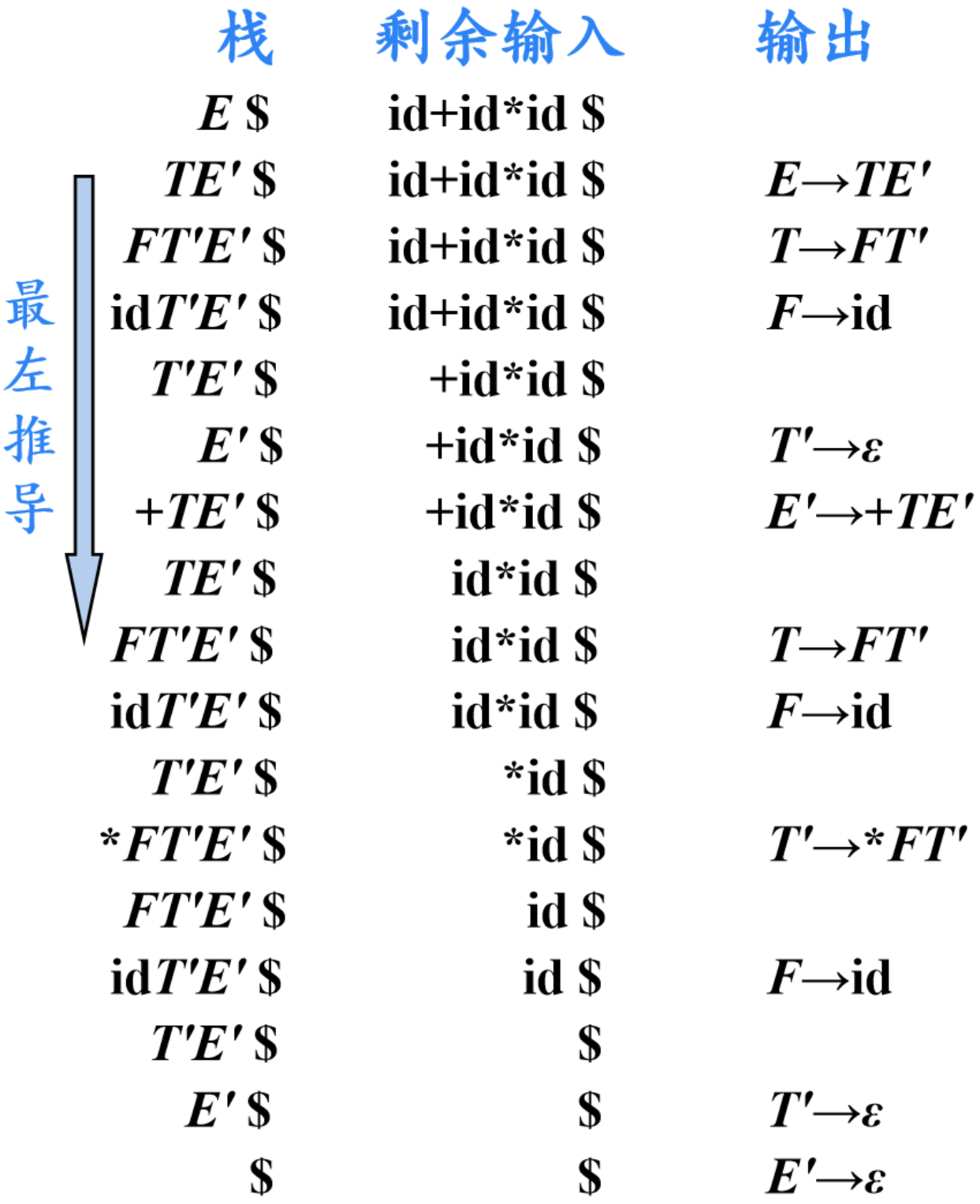
4.2.3 非递归的预测分析法

- 非递归的预测分析**显式**地维护一个**栈结构**，而不是通过递归调用的方式**隐式**地维护栈。这样的语法分析器可以模拟**最左推导**过程



例

非终结符	输入符号					
	id	+	*	(	)	$
E	$E\to TE'$			$E\to TE'$		
E'		$E'\to +TE'$			$E'\to\varepsilon$	$E'\to\varepsilon$
T	$T\to FT'$			$T\to FT'$		
T'		$T'\to\varepsilon$	$T'\to *FT'$		$T'\to\varepsilon$	$T'\to\varepsilon$
F	$F\to$ id			$F\to(E)$		


栈
剩余输入
输出
最左推导
E $
id+id*id $
TE' $
id+id*id $
E→TE'
FT'E' $
id+id*id $
T→FT'
idT'E' $
id+id*id $
F→id
T'E' $
+id*id $
E' $
+id*id $
T'→ε
+TE' $
+id*id $
E'→+TE'
TE' $
id*id $
FT'E' $
id*id $
T→FT'
idT'E' $
id*id $
F→id
T'E' $
*id $
*FT'E' $
*id $
T'→*FT'
FT'E' $
id $
idT'E' $
id $
F→id
T'E' $
$
E' $
$
T'→ε
$
$
E'→ε

表驱动的预测分析法

- 输入：一个串w和文法G的分析表M
- 输出：如果w在$L(G)$中，输出w的最左推导；否则给出错误指示
- 方法：最初，语法分析器的格局如下：输入缓冲区中是$w\$$，G的开始符号位于栈顶，其下面是$\$$。下面的程序使用预测分析表M生成了处理这个输入的预测分析过程

```
设置ip使它指向w的第一个符号，其中ip 是输入指针;
令X=栈顶符号;
while (X≠ $ ) {    /* 栈非空 */
        if ( X 等于ip所指向的符号a) 执行栈的弹出操作，将ip向前移动一个位置;
        else if (X是一个终结符号) error ( ) ;
        else if ( M[X，a]是一个报错条目) error ( ) ; /*预测表中没有对应条目*/
        else if ( M[X，a] = X→Y1Y2 … Yk ) {
                输出产生式X→Y1Y2 … Yk ;
                弹出栈顶符号X;
                将Yk，Yk-1 …，Y1 压入栈中，其中Y1位于栈顶。
        }
        令X=栈顶符号
}
```

递归的预测分析法vs.非递归的预测分析法

	递归的预测分析法	非递归的预测分析法
程序规模	程序规模较大，不需载入分析表	主控程序规模较小，需载入分析表（表较小）
直观性	较好	较差
效率	较低	分析时间大约正比于待分析程序的长度
自动生成	较难	较易

预测分析法实现步骤

1）构造文法　　无二义性

2）改造文法：消除二义性、消除左递归、消除回溯

3）求每个变量的*FIRST*集和*FOLLOW*集，从而求得每个候选式的*SELECT*集

确定性

4）检查是不是*LL*(1) 文法。若是，构造预测分析表

5）对于递归的预测分析，根据预测分析表为每一个非终结符编写一个过程；对于非递归的预测分析，实现表驱动的预测分析算法

无二义性vs.确定性

4.2.4 预测分析中的错误检测

- 两种情况下可以**检测到错误**
  - 栈顶的**终结符**和当前输入符号不匹配
  - 栈顶**非终结符**与当前输入符号在预测分析表**对应项中的信息为空**

- 预测分析中的**错误恢复**

➢ **恐慌模式（Panic Mode）**

➢ 忽略输入中的一些符号，直到输入中出现由设计者选定的同步词法单元(*synchronizing token*)集合中的某个词法单元

➢ 其效果依赖于同步集合的选取。集合的选取应该使得语法分析器能从实际遇到的错误中快速恢复　非终结符在栈顶

➢ 例如可以把*FOLLOW(A)*中的所有终结符放入非终结符*A*的同步记号集合

➢ 如果终结符在栈顶而不能匹配，一个简单的办法就是弹出此终结符

例

非终结符	输入符号					
	id	+	*	(	)	$
E	*E*→*TE′*			*E*→*TE′*	synch	synch
E′		*E′*→+*TE′*			*E′*→ε	*E′*→ε
T	*T*→*FT′*	synch		*T*→*FT′*	synch	synch
T′		*T′*→ε	*T′*→**FT′*		*T′*→ε	*T′*→ε
F	*F*→id	synch	synch	*F*→(*E*)	synch	synch

X	FOLLOW(*X*)
E	$)
E′	$)
T	+) $
T′	+) $
F	* +) $

*Synch*表示根据相应非终结符的*FOLLOW*集得到的同步词法单元

➢ 分析表的使用方法

➢ 如果M[*A*,*a*]是空，表示检测到错误，根据恐慌模式，忽略输入符号*a*

➢ 如果M[*A*,*a*]是*synch*，则弹出栈顶的非终结符*A*，试图继续分析后面的语法成分

➢ 如果栈顶的终结符和输入符号不匹配，则弹出栈顶的终结符

非终结符	输入符号					
	id	+	*	(	)	$
E	$E\to TE'$			$E\to TE'$	synch	synch
E'		$E'\to +TE'$			$E'\to\varepsilon$	$E'\to\varepsilon$
T	$T\to FT'$	synch		$T\to FT'$	synch	synch
T'		$T'\to\varepsilon$	$T'\to *FT'$		$T'\to\varepsilon$	$T'\to\varepsilon$
F	$F\to$id	synch	synch	$F\to(E)$	synch	synch

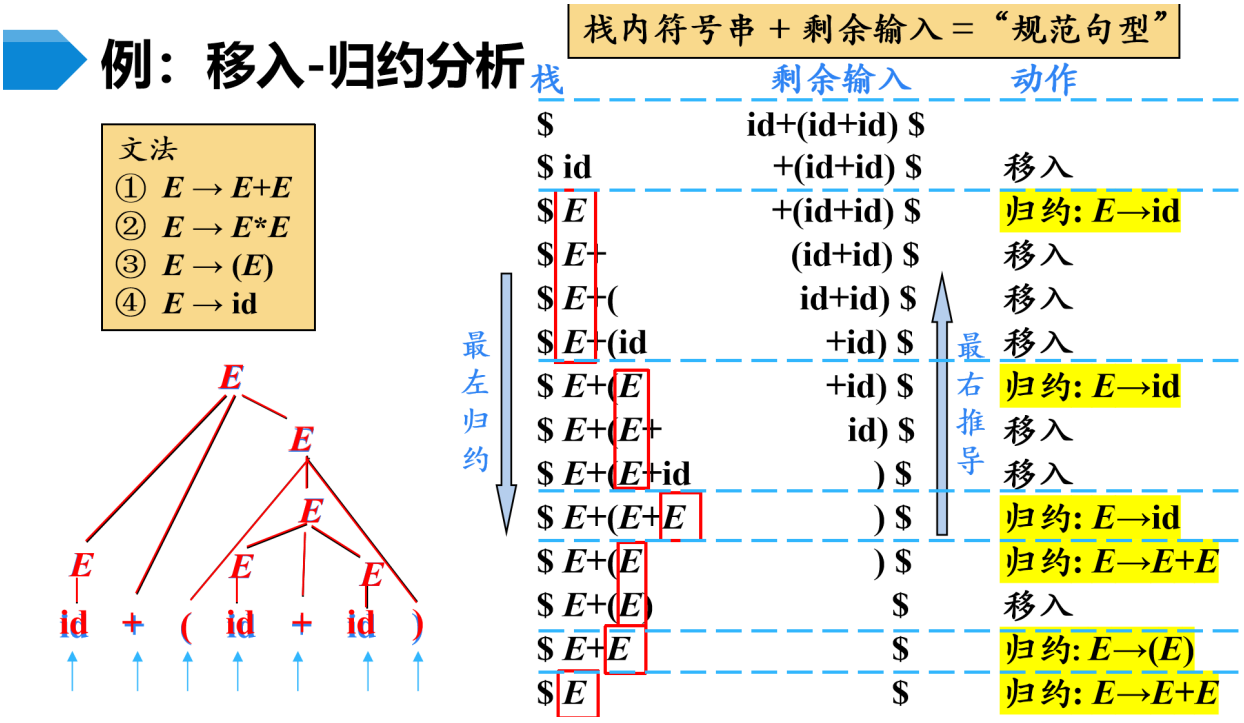
栈	剩余输入	
E $	+id*+id $	ignore +
E $	id*+id $	
TE' $	id*+id $	
$FT'E'$ $	id*+id $	
id$T'E'$ $	id*+id $	
$T'E'$ $	*+id $	
*$FT'E'$ $	*+id $	
$FT'E'$ $	+id $	error
$T'E'$ $	+id $	
E' $	+id $	
+TE' $	+id $	
TE' $	id $	
$FT'E'$ $	id $	
id$T'E'$ $	id $	
$T'E'$ $		$
E' $	$	
$	$	

4.3 自底向上的分析

- 从分析树的底部(叶节点)向顶部(根节点)方向构造分析树
- 可以看成是将输入**串w归约为文法开始符号S**的过程
- 自顶向下的语法分析采用最左推导方式（构造句子的最左推导）
- 自底向上的语法分析采用最左归约方式（反向构造句子的最右推导）
- 自底向上语法分析的通用框架
  - **移入-归约分析**(Shift-Reduce Parsing)

4.3.1 移入-归约分析

例：移入-归约分析



- 每次归约的符号串称为**“句柄”**

  eg.归约：E→id，句柄：id

- **移入-归约分析的工作过程**
  - 在对**输入串**的一次从左到右扫描过程中，语法分析器将**零个或多个**输入符号移入到**栈的顶端**，**直到**它可以对栈顶的一个文法符号串β进行**归约为止**
  - 将β归约为某个产生式的左部
  - 语法分析器不断地重复这个循环，直到它检测到一个**语法错误**，或者**栈中包含了开始符号且输入缓冲区为空**(当进入这样的格局时，语法分析器停止运行，并宣称成功完成了语法分析)为止
- **移入-归约分析器可采取的4种动作**
  - **移入**：将下一个输入符号移到栈的顶端
  - **归约**：被归约的符号串的右端必然处于栈顶。语法分析器在栈中确定这个串的左端，并决定用哪个非终结符来替换这个串
  - **接收**：宣布语法分析过程成功完成
  - **报错**：发现一个语法错误，并调用错误恢复子例程

- **移入-归约分析中存在的问题**
  - 归约-归约冲突
  - 移入-归约冲突
- **归约-归约冲突**

例：

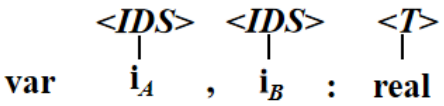
(1) <*S*>→var <*IDS*> : <*T*>

(2) <*IDS*>→i

(3) <*IDS*>→<*IDS*> , i

(4) <*T*>→real | int

栈	剩余输入	动作
$	var i_A , i_B : real $	
$ var	i_A , i_B : real $	移入
$ var i_A	, i_B : real $	移入
$ var <*IDS*>	, i_B : real $	归约
$ var <*IDS*> ,	i_B : real $	移入
$ var <*IDS*> , i_B	: real $	移入
$ var <*IDS*> , <*IDS*>	: real $	归约
$ var <*IDS*> , <*IDS*> :	real $	移入
$ var <*IDS*> , <*IDS*> : real	$	移入
$ var <*IDS*> , <*IDS*> :<*T*>	$	归约



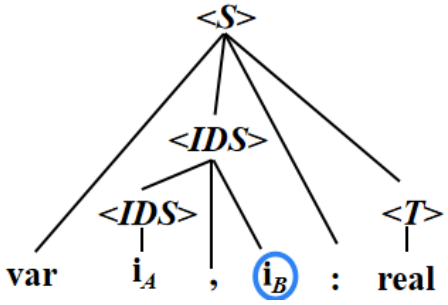
例：

(1) <*S*>→var <*IDS*> : <*T*>

(2) <*IDS*>→i

(3) <*IDS*>→<*IDS*> , i

(4) <*T*>→real | int



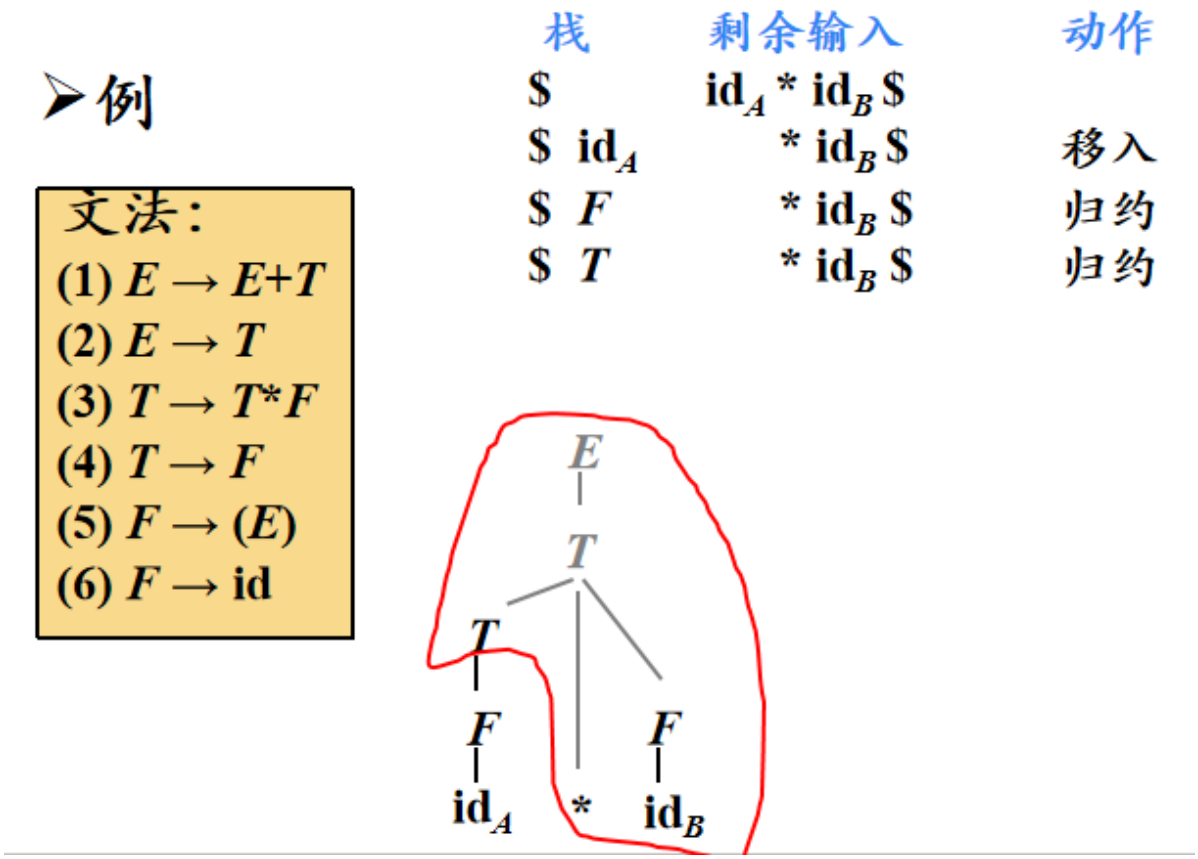
造成错误的原因：
错误地识别了句柄

栈	剩余输入	动作
$	var i_A , i_B : real $	
$ var	i_A , i_B : real $	移入
$ var i_A	, i_B : real $	移入
$ var <*IDS*>	, i_B : real $	归约
$ var <*IDS*> ,	i_B : real $	移入
$ var <*IDS*> , i_B	: real $	移入
$ var <*IDS*>	: real $	归约
$ var <*IDS*> :	real $	移入
$ var <*IDS*> : real	$	移入
$ var <*IDS*> :<*T*>	$	归约
$ <*S*>	$	归约

句柄：句型的最左直接短语

- 每个子树的边缘都可以称为**短语**
- 高度为2的子树边缘是**直接短语**
- 短语←句型
- **正确的识别句柄**
  - 归约的是句型的**最左直接短语（最右句型的句柄）**

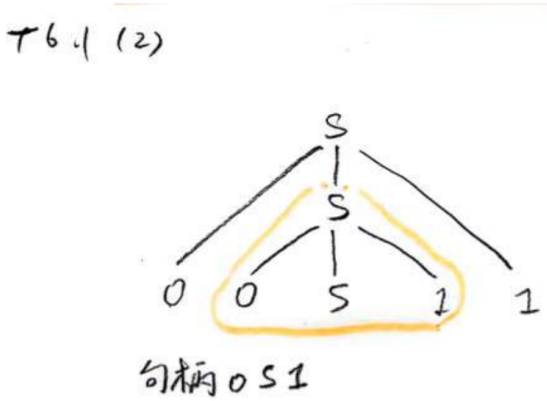
- **移入-归约冲突**



习题

➢对于文法S → 0 S 1 | 0 1，指出下面各个最右句型的句柄：

➢(2) 00S11

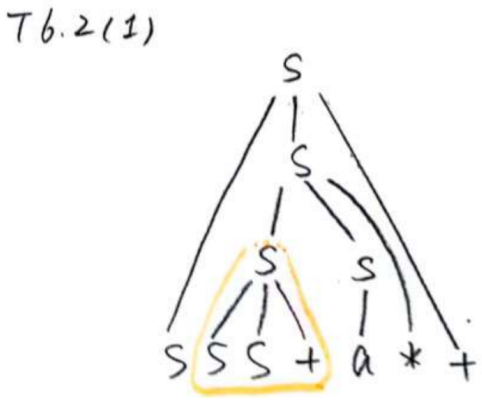


• T 6.2 (1)

➢对于文法S → S S + | S S * | a，指出下面各个最右句型的句柄：

➢(1) SSS+a*+

短语、直接短语、句柄



注：最右句型的句柄就是最左直接短语（第一个归约的短语）

4.4 LR分析法

- *LR*文法(*Knuth*, 1963)是最大的、可以构造出相应移入-归约语法分析器的文法类
  - *L*: 对输入进行从左到右的扫描
  - *R*: 反向构造出一个最右推导序列
- *LR*(*k*)分析
  - （决定是否归约时）需要向前查看*k*个输入符号的*LR*分析

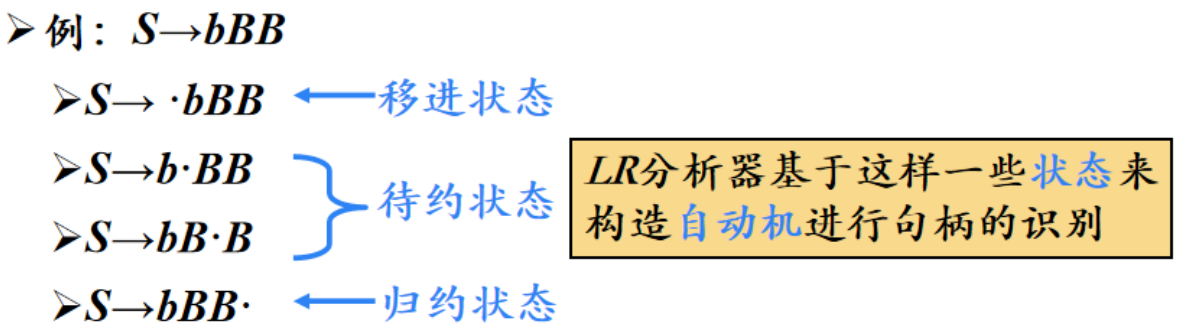
> *k* = 0 和 *k* = 1 这两种情况具有实践意义
> 当省略(*k*)时，表示*k* =1

💡 **LL(1)文法：**

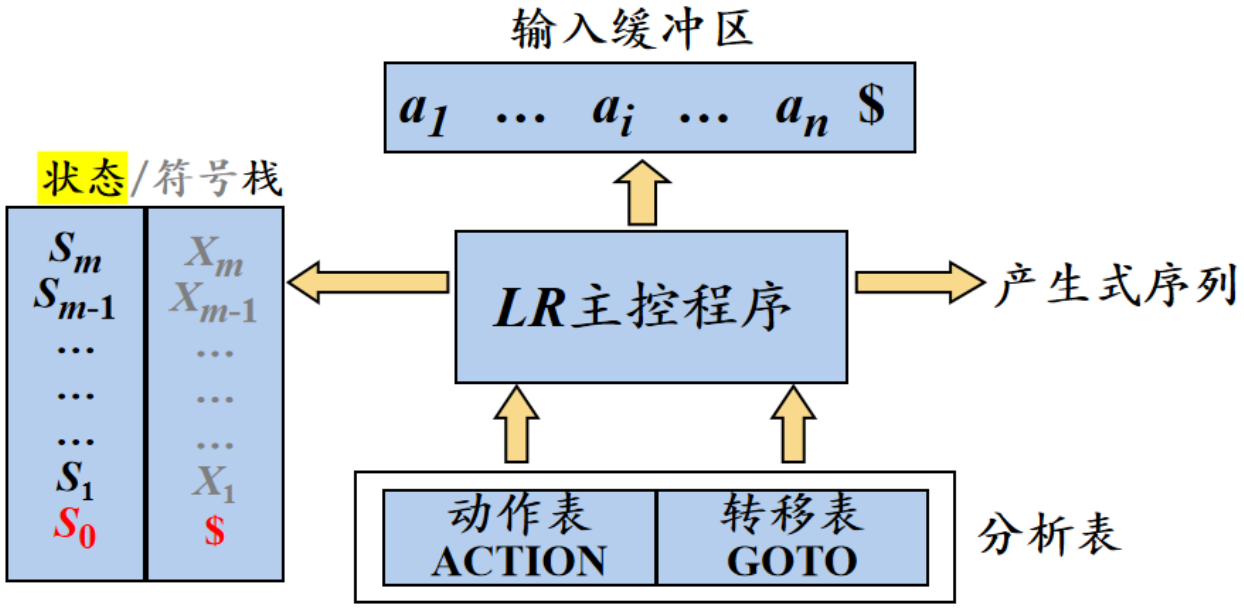
> - 第一个“*L*”表示从左（Left）向右扫描输入
> - 第二个“*L*”表示产生最左（Left）推导
> - “1”表示在每一步中只需要向前看一个输入符号来决定语法分析动作

LR 分析法的基本原理

- 自底向上分析的**关键问题**是什么？
  - 如何**正确地识别句柄**
- 句柄是**逐步形成**的，用**“状态”**表示**句柄识别的进展程度**



LR 分析器（自动机）的总体结构



符号状态栈同进同出

LR分析表操作

- 例
  - 文法
    - ① $S \rightarrow BB$
    - ② $B \rightarrow aB$
    - ③ $B \rightarrow b$

sn：将符号a、状态n压入栈
rn：用第n个产生式进行归约

状态	ACTION			GOTO	
	a	*b*	$	*S*	*B*
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

输入 *b a b*

栈	剩余输入
0 4	
$*b*	*a b* $

B
|
输入 *b a b*

栈	剩余输入
0	
$	*a b* $

B
|
输入 *b a b*

栈	剩余输入
0	
$*B*	*a b* $

B
|
输入 *b a b*

栈	剩余输入
0 2	
$*B*	*a b* $

$$s_i \xrightarrow[A\,(归约)]{a\,(移入)} s_j$$

LR 分析器的工作过程

➢初始化

s_0	
\$	$a_1 a_2 \ldots a_n$ \$

➢一般情况下

$s_0 s_1 \ldots s_m$	
$\$X_1 \ldots X_m$	$a_i a_{i+1} \ldots a_n$ \$

①如果ACTION $[s_m, a_i]$= **sx**，那么格局变为：

$s_0 s_1 \ldots s_m x$	
$\$X_1 \ldots X_m a_i$	$a_{i+1} \ldots a_n$ \$

②如果ACTION$[s_m, a_i]$= **rx** 表示用第x个产生式$A \rightarrow X_{m-(k-1)} \ldots X_m$

进行归约，那么格局变为：

$s_0 s_1 \ldots s_{m-k}$	
$\$ X_1 \ldots X_{m-k} A$	$a_i a_{i+1} \ldots a_n$ \$

如果**GOTO**$[s_{m-k}, A]$=y，那么格局变为：

$s_0 s_1 \ldots s_{m-k}\ y$	
$\$ X_1 \ldots X_{m-k} A$	$a_i a_{i+1} \ldots a_n$ \$

③如果ACTION[s_m, a_i]=*acc*，那么分析成功

④如果ACTION[s_m, a_i]=*err*，那么出现语法错误

LR分析算法

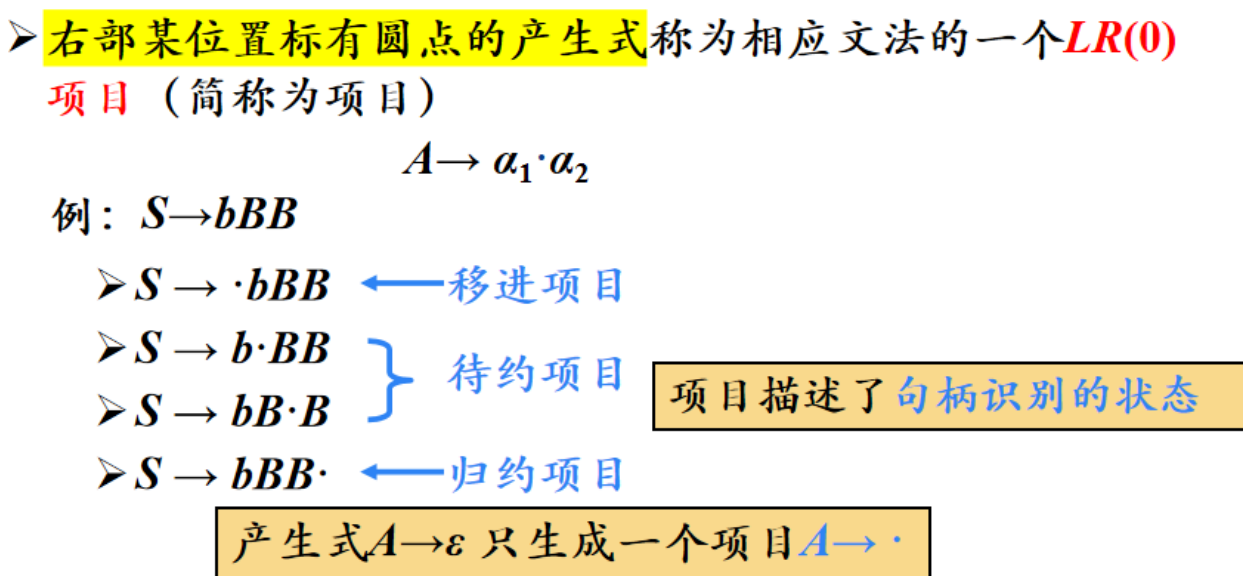
- 输入：串w和LR语法分析表，该表描述了文法G的ACTION函数和GOTO函数。
- 输出：如果w在$L(G)$中，则输出w的自底向上语法分析过程中的归约步骤；否则给出一个错误指示。
- 方法：初始时，语法分析器栈中的内容为初始状态s_0，输入缓冲区中的内容为$w$$。然后，语法分析器执行下面的程序：

```
令a为w$的第一个符号;
while(1) { /* 永远重复*/
    令s是栈顶的状态;
    if ( ACTION [s, a] = st ) {
        将t压入栈中;
        令a为下一个输入符号;
    } else if (ACTION [s, a] =归约A → β ) {
        从栈中弹出 | β | 个符号;
        将GOTO [t, A]压入栈中;
        输出产生式 A → β ;
    } else if (ACTION [s, a] =接受) break;  /* 语法分析完成*/
    else调用错误恢复例程;
}
```

如何构造给定文法的LR分析表？

- LR(0)分析
- SLR分析
- LR(1)分析
- LALR分析

4.4.1 LR(0)分析



- 一个（ 项目 ）指明了在LR分析过程中的某个时刻所能看到产生式多大一部分

增广文法

➢ 如果G是一个以S为开始符号的文法，则G的增广文法G'就是在G中加上新开始符号S'和产生式$S' \to S$而得到的文法

➢ 例

1) $E \to E + T$
2) $E \to T$
3) $T \to T * F$
4) $T \to F$
5) $F \to (E)$
6) $F \to \text{id}$

0) $E' \to E$
1) $E \to E + T$
2) $E \to T$
3) $T \to T * F$
4) $T \to F$
5) $F \to (E)$
6) $F \to \text{id}$

引入这个新的开始产生式的目的是使得文法开始符号仅出现在一个产生式的左边，从而使得分析器只有一个接受状态

文法中的项目

① $S' \to S$　② $S \to \text{v}I:T$　③ $I \to I,\text{i}$　④ $I \to \text{i}$　⑤ $T \to \text{r}$

初始项目

(0) $S' \to \cdot S$
(1) $S' \to S \cdot$
(2) $S \to \cdot \text{v}I:T$
(3) $S \to \text{v} \cdot I:T$
(4) $S \to \text{v}I \cdot :T$
(5) $S \to \text{v}I: \cdot T$
(6) $S \to \text{v}I:T \cdot$
(7) $I \to \cdot I,\text{i}$
(8) $I \to I \cdot ,\text{i}$
(9) $I \to I, \cdot \text{i}$
(10) $I \to I,\text{i} \cdot$
(11) $I \to \cdot \text{i}$
(12) $I \to \text{i} \cdot$
(13) $T \to \cdot \text{r}$
(14) $T \to \text{r} \cdot$

归约项目

接收项目

➢ 后继项目 (*Successive Item*)

➢ 同属于一个产生式的项目，但圆点的位置只相差一个符号，则称后者是前者的后继项目

➢ $A \to \alpha \cdot X\beta$的后继项目是$A \to \alpha X \cdot \beta$

项目集闭包

可以把等价的项目组成一个项目集(I)，称为项目集闭包(*Closure of Item Sets*)，每个项目集闭包对应着自动机的一个状态

- 点后面有**非终结符**，找到这个非终结符的产生式，点在右部最左边的项目是**等价项目**

(0) $S'\rightarrow\cdot S$
(1) $S'\rightarrow S\cdot$
(2) $S\rightarrow\cdot vI{:}T$
(3) $S\rightarrow v\cdot I{:}T$
(4) $S\rightarrow vI\cdot{:}T$
(5) $S\rightarrow vI{:}\cdot T$
(6) $S\rightarrow vI{:}T\cdot$
(7) $I\rightarrow\cdot I,i$
(8) $I\rightarrow I\cdot,i$
(9) $I\rightarrow I,\cdot i$
(10) $I\rightarrow I,i\cdot$
(11) $I\rightarrow\cdot i$
(12) $I\rightarrow i\cdot$
(13) $T\rightarrow\cdot r$
(14) $T\rightarrow r\cdot$

➢ 计算给定项目集I的闭包

$$CLOSURE(I) = I \cup \{B\rightarrow\cdot\gamma \mid A\rightarrow\alpha\cdot B\beta\in CLOSURE(I), B\rightarrow\gamma\in P\}$$

```
SetOfItems CLOSURE (I) {
    J = I;
    repeat
        for (J中的每个项A → α·Bβ)
            for (G的每个产生式B → γ)
                if (项B → ·γ不在J中)
                    将B → ·γ加入J中;
    until 在某一轮中没有新的项被加入到J中;
    return J;
}
```

GOTO（）函数

➢返回项目集I对应于文法符号X的后继项目集闭包

$$GOTO(I,X)=CLOSURE(\{A\rightarrow\alpha X\cdot\beta \mid A\rightarrow\alpha\cdot X\beta\in I\})$$

```
SetOfItems GOTO (I, X) {
        将J 初始化为空集;
        for (I 中的每个项A → α·Xβ)
                将项 A → αX·β 加入到集合J 中;
        return CLOSURE (J);
}
```

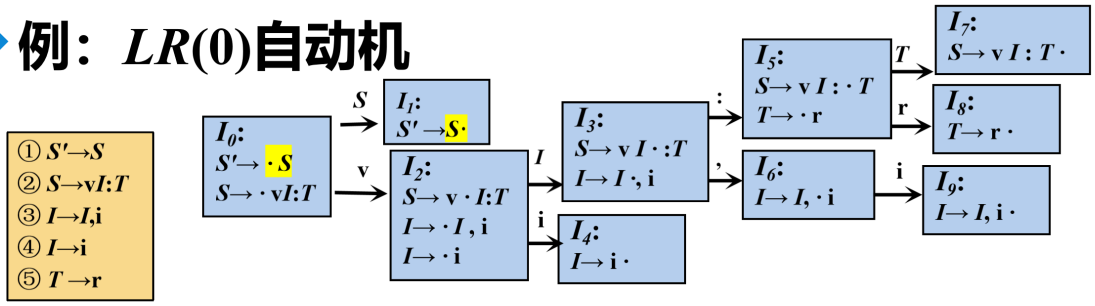
构造LR(0)自动机的状态集

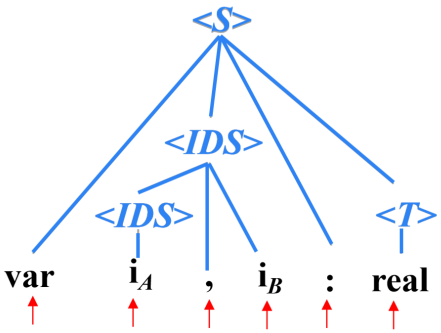
➢规范$LR(0)$ 项集族(*Canonical LR(0) Collection*)

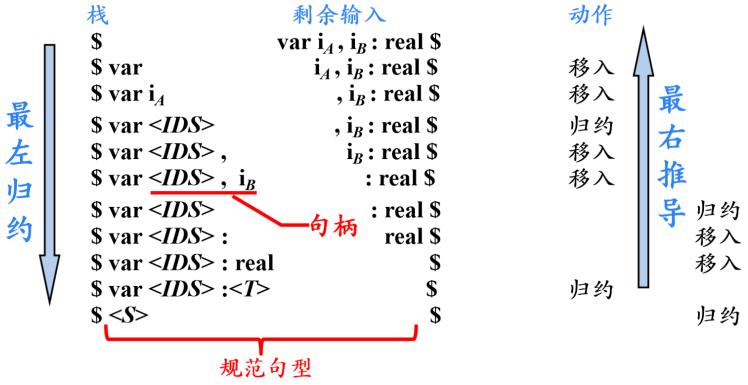
$$C=\{I_0\}\cup\{I \mid \exists J\in C, X\in V_N\cup V_T, I=GOTO(J,X)\}$$

```
void items( G' ) {
        C={ CLOSURE ({[ S'→ ·S ]}) };
        repeat
                for (C中的每个项集 I)
                        for(每个文法符号X)
                                if ( GOTO (I, X)非空且不在C中)
                                        将GOTO (I, X)加入C中;
        until在某一轮中没有新的项集被加入到C中;
}
```

例：LR(0)自动机





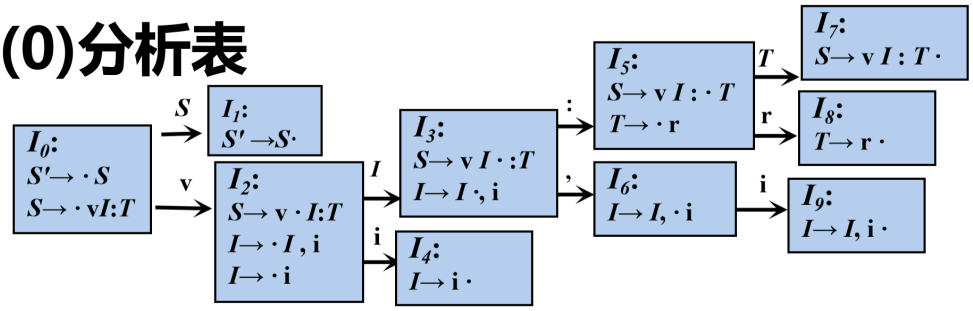


- 第一次归约时，进入 I4，弹出 iA，并且自动机进入上一个状态，即I2，压入IDS，进入到I3

构造LR（0）分析表

构造LR(0)分析表

① $S'\to S$
② $S\to vI{:}T$
③ $I\to I,i$
④ $I\to i$
⑤ $T\to r$



状态	ACTION						GOTO		
	v	:	,	i	r	$	S	I	T
0	s2						1		
1						acc			
2				s4				3	
3		s5	s6						
4	r4	r4	r4	r4	r4	r4			
5					s8				7
6				s9					
7	r2	r2	r2	r2	r2	r2			
8	r5	r5	r5	r5	r5	r5			
9	r3	r3	r3	r3	r3	r3			

信息为空的项目都设置为“*error*”

- CLOSURE ({[$S'\to \cdot S$] })$\to C$ **C：自动机状态集合**
- for each $I_i \in C$
  - if $A\to\alpha\cdot a\beta\in I_i$: 令I_j=*GOTO*(I_i, a); *ACTION*[*i*, a]=s_j
  - if $A\to\alpha\cdot B\beta\in I_i$:令I_j=*GOTO*(I_i, B); *GOTO*[*i*, B]=*j*
  - if $A\to\alpha\cdot\in I_i$: { if $A=S'$ ($S'\to S\cdot\in I_i$) *ACTION* [*i*, $]=*acc*
    if $A\neq S'$ for $\forall a\in V_T\cup\{\$\}$ do *ACTION*[*i*, a]=r_j（*j*是产生式$A\to\alpha$的编号）
- 没有定义的所有条目都设置为“*error*”

LR(0) 自动机的形式化定义

➢ 文法

$$G=(V_N, V_T, P, S)$$

➢ $LR(0)$ 自动机

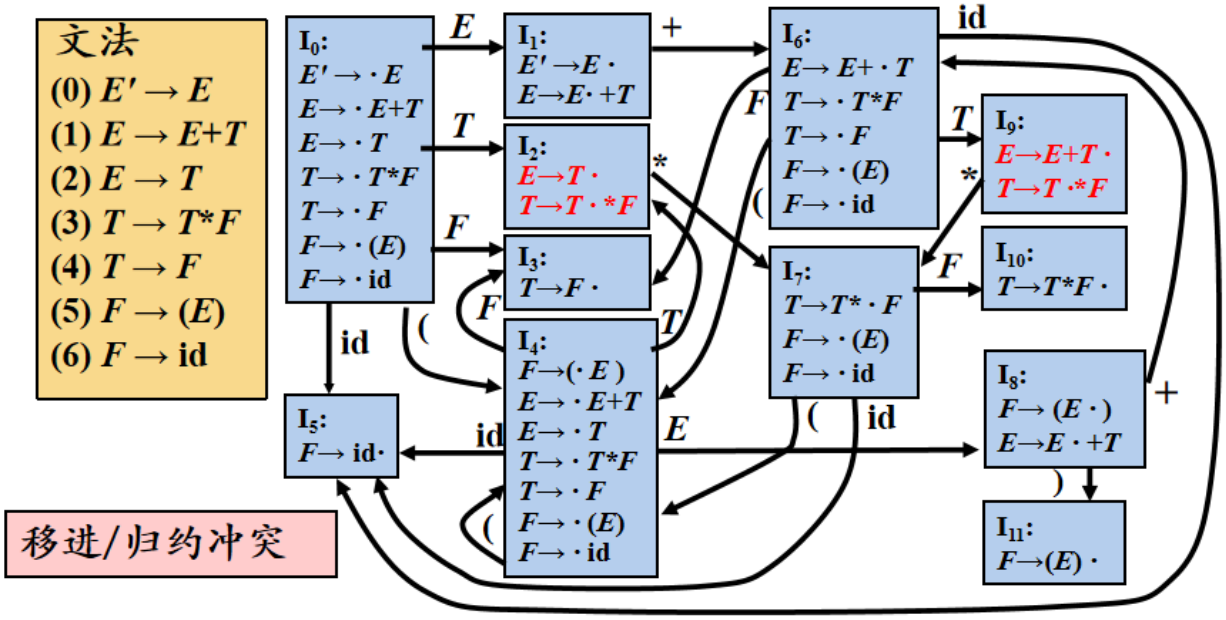
$$M=(C, V_N \cup V_T, GOTO, I_0, F)$$

➢ $C=\{I_0\} \cup \{I \mid \exists J \in C, X \in V_N \cup V_T, I=GOTO(J,X)\}$

➢ $I_0=CLOSURE(\{S' \rightarrow .S\})$

➢ $F=\{CLOSURE(\{S' \rightarrow S.\})\}$

LR(0) 分析过程中的冲突

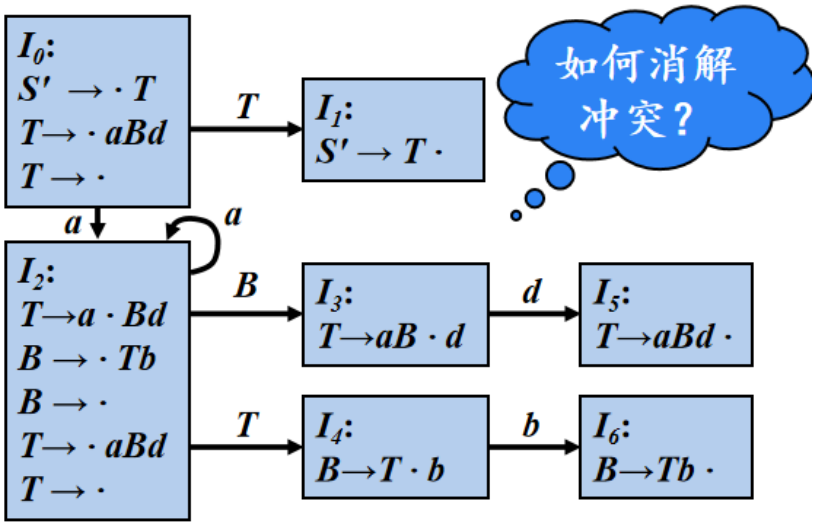


状态	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s_4			1	2	3
1		s6				acc			
2	r2	r2	r2/s7	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4	s5			s4			8	2	3
5	r6	r6	r6	r6	r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9	r1	r1	r1/s7	r1	r1	r1			
10	r3	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5	r5			

例：移进/归约冲突和归约/归约冲突

文法
(0) $S' \rightarrow T$
(1) $T \rightarrow aBd$
(2) $T \rightarrow \varepsilon$
(3) $B \rightarrow Tb$
(4) $B \rightarrow \varepsilon$

如果LR(0)分析表中没有语法分析动作冲突，那么给定的文法就称为LR(0)文法

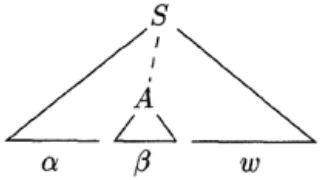


不是所有CFG都能用LR(0)方法进行分析，也就是说，CFG不总是LR(0)文法

4.4.2 SLR分析

LR(0)自动机为什么消解不了某些冲突？

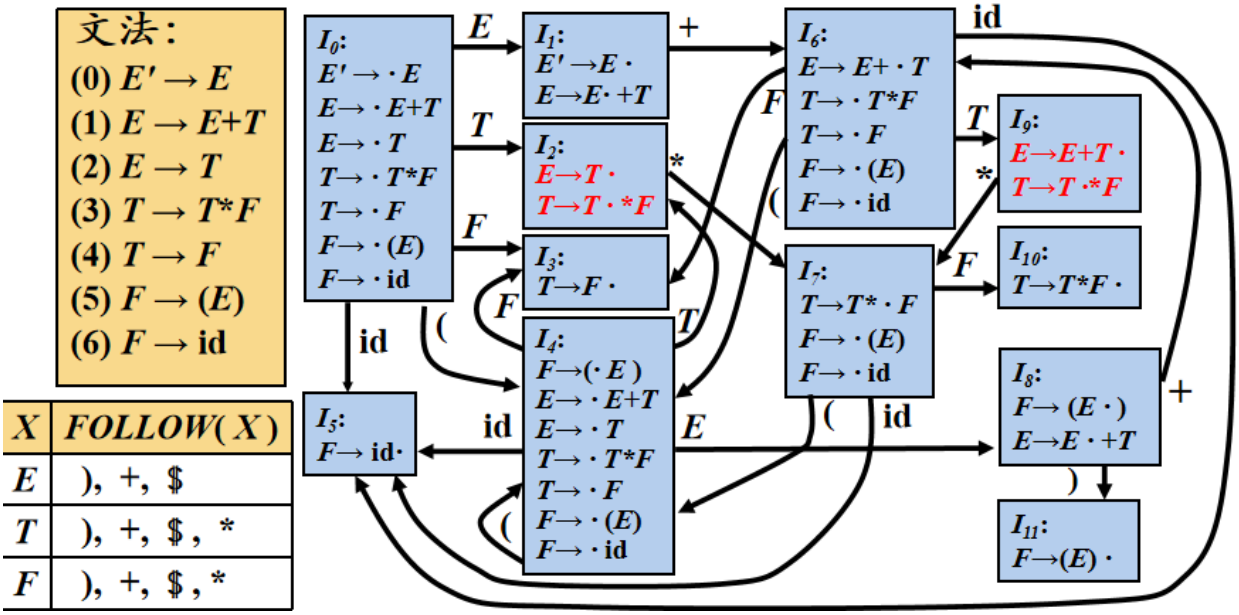
- 句柄都是相对一个句型而言的，因此应该将句柄的识别放在句型这样一个上下文环境中考虑
- 对于$A\to\beta$，是否应该将β归约为A取决于当前句型对应的分析树中A所处的上下文环境



- $LR(0)$考虑了A的上文（规范句型的前缀），但未考虑A的下文，因此消解冲突能力有限

SLR分析

往前看一个符号



➢ *SLR*分析法的基本思想

已知项目集 I:

$A_1 \rightarrow \alpha_1.a_1\beta_1$

$A_2 \rightarrow \alpha_2.a_2\beta_2$

…

$A_m \rightarrow \alpha_m.a_m\beta_m$

} m个移进项目

$B_1 \rightarrow \gamma_1.$

$B_2 \rightarrow \gamma_2.$

…

$B_n \rightarrow \gamma_n.$

} n个归约项目

如果集合$\{a_1, a_2, \ldots, a_m\}$和 $FOLLOW(B_1)$，$FOLLOW(B_2)$，…，$FOLLOW(B_n)$两两不相交，则项目集I中的冲突可以按以下原则解决：

设a是下一个输入符号

- 若$a \in \{a_1, a_2, \ldots, a_m\}$，则移进a
- 若$a \in FOLLOW(B_i)$，则用产生式$B_i \rightarrow \gamma_i$归约
- 此外，报错

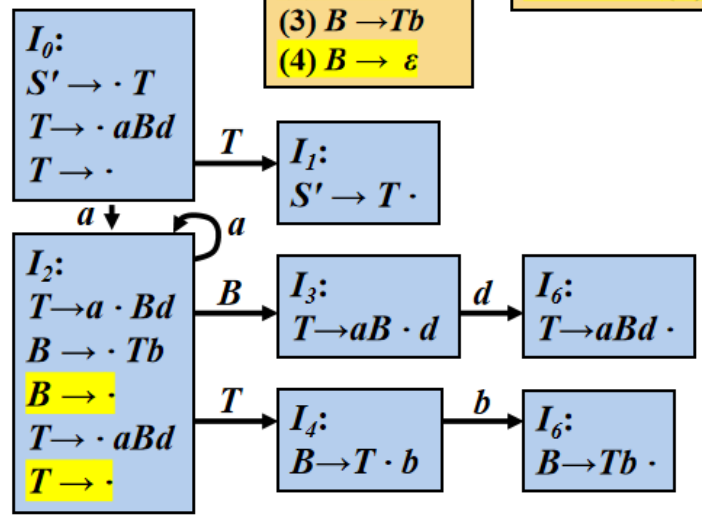
表达式文法的*SLR*分析表

状态	*ACTION*						*GOTO*		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

例

文法
(0) $S'\rightarrow T$
(1) $T\rightarrow aBd$
(2) $T\rightarrow \varepsilon$
(3) $B\rightarrow Tb$
(4) $B\rightarrow \varepsilon$

$FOLLOW(S')=\{\$\}$
$FOLLOW(T)=\{\$, b\}$
$FOLLOW(B)=\{d\}$



SLR(1)分析表

状态	ACTION				GOTO	
	a	b	d	$	T	B
0	s2	r2		r2	1	
1				acc		
2	s2	r2	r4	r2	4	3
3			s5			
4		s6				
5		r1		r1		
6			r3			

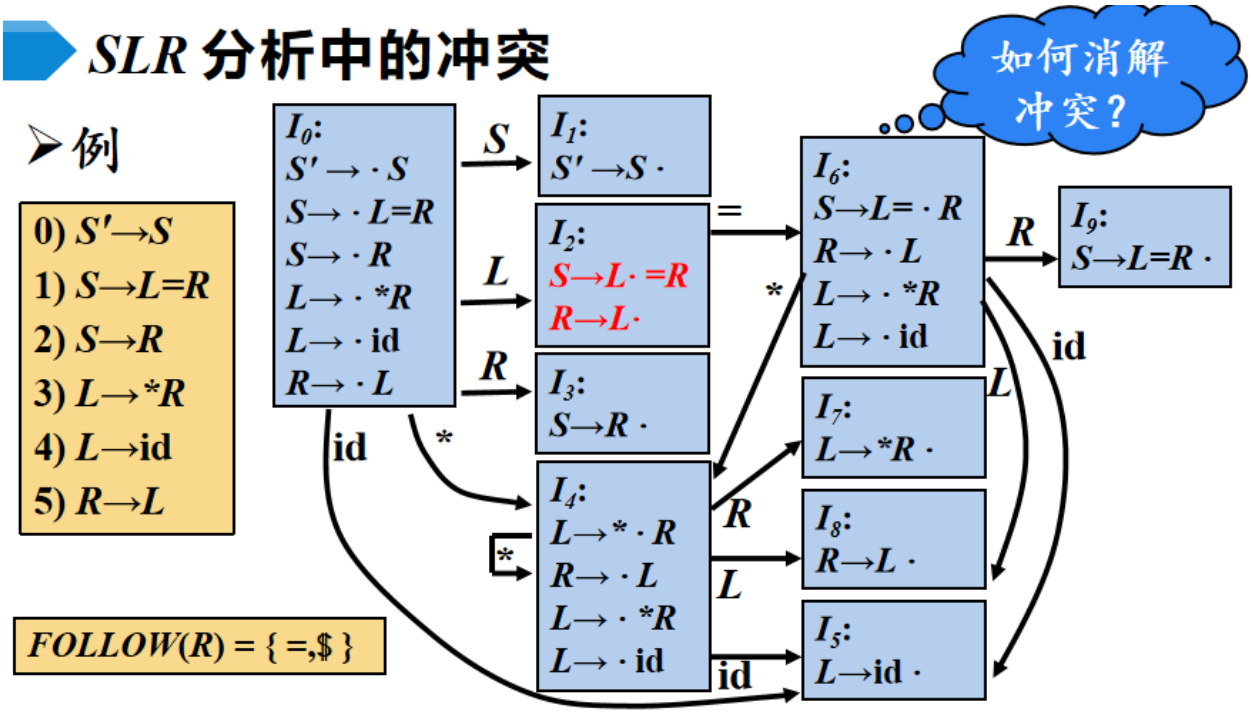
SLR 分析表构造算法

- 构造G'的规范$LR(0)$项集族$C=\{I_0, I_1, \dots, I_n\}$。
- 根据I_i构造得到状态i。状态i的语法分析动作按照下面的方法决定：
  - if $A\rightarrow \alpha\cdot a\beta \in I_i$ and $GOTO(I_i, a)=I_j$ then $ACTION[i, a]=sj$
  - if $A\rightarrow \alpha\cdot B\beta \in I_i$ and $GOTO(I_i, B)=I_j$ then $GOTO[i, B]=j$
  - if $A\rightarrow \alpha\cdot \in I_i$ 且 $A\neq S'$ then for $\forall a\in FOLLOW(A)$ do
    $ACTION[i, a]=rj$ （j是产生式$A\rightarrow\alpha$的编号）
  - if $S'\rightarrow S\cdot \in I_i$ then $ACTION[i, \$]=acc$;
- 没有定义的所有条目都设置为“error”。

如果给定文法的SLR分析表中不存在有冲突的动作，
那么该文法称为SLR文法

SLR分析中也有冲突

SLR 分析中的冲突



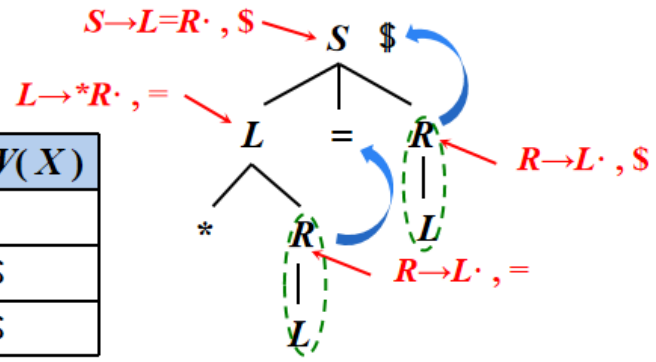
4.4.3 LR(1)分析

- SLR分析存在的问题
  - SLR只是简单地考察下一个输入符号b是否属于与归约项目A→α相关联的FOLLOW(A)，但b∈FOLLOW(A)只是归约α的一个必要条件，而非充分条件

➢对于产生式$A\to\alpha$的归约，在不同的使用位置，A会要求不同的后继符号

0) $S'\to S$
1) $S\to L=R$
2) $S\to R$
3) $L\to *R$
4) $L\to$ id
5) $R\to L$

X	$FOLLOW(X)$
S	\$
L	=, \$
R	=, \$



➢在特定位置，A的后继符集合是$FOLLOW(A)$的子集

规范LR（1）项目

➢将一般形式为 $[A\to\alpha\cdot\beta, a]$的项称为 $LR(1)$ 项，其中$A\to\alpha\beta$ 是一个产生式，a 是一个终结符(这里将\$视为一个特殊的终结符)它表示在当前状态下，A后面必须紧跟的终结符，称为该项的展望符(*lookahead*)

- ➢$LR(1)$ 中的1指的是项的第二个分量的长度
- ➢在形如$[A\to\alpha\cdot\beta, a]$且$\beta\neq\varepsilon$的项中，展望符a没有任何作用
- ➢但是一个形如$[A\to\alpha\cdot, a]$的项在只有在下一个输入符号等于a时才可以按照$A\to\alpha$ 进行归约
  - ➢这样的a的集合总是$FOLLOW(A)$的子集，而且它通常是一个真子集

LR(1)自动机某项集中的项目$[A\to\alpha\cdot X\beta, b]$若有后继项目$A\to\alpha X\cdot\beta$，其后继项目的展望符是：

A 继承的(传播的) b

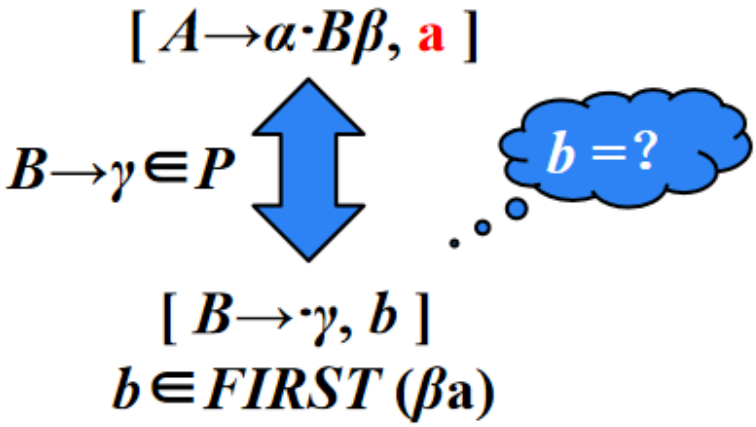
B FOLLOW(X)

C FOLLOW(A)

D **FIRST(βb)**

- **注：这个不是算等价的项目不需要用到下面的，即不是计算X→... 的展望符，他同样是计算A的展望符，根据展望符的定义，A后面必须紧跟的终结符仍是b**

等价LR（1）项目



当$\beta \Rightarrow^{+} \varepsilon$时，此时$b$=a叫继承的后继符，否则叫自生的后继符

2. **项集闭包计算**

   - 如果在项集中存在一个项 $A \to \alpha \cdot B\beta, a$ 并且 B 是一个非终结符，则需要把 B 的产生式加入到项集中，但需要计算这些新项目的展望符。

3. **展望符传播规则**

   - 对于项目 $A \to \alpha \cdot B\beta, a$，假设 B → γ 是一个产生式，新产生的项目为 $B \to \cdot \gamma, ?$。此时，展望符的确定分以下几种情况：

   规则1：FIRST集传播

   - 如果存在 $A \to \alpha \cdot B\beta, a$，且产生项目 $B \to \cdot \gamma, ?$，那么新项目的展望符为 `FIRST(βa)`。
   - 具体来说，新展望符集合为 `FIRST(βa)`，即计算从 β 开始以及接着的展望符 a 的所有可能的第一个符号。

   规则2：继承的展望符

   - 如果 β 可以推导为空串 ε（即 β ⇒* ε），则新项目的展望符应包括 `a`。
   - 简言之，当 β 可以为空时，原项目的展望符直接传播给新项目。

详细解释和例子

例子1：单个符号的传播

考虑产生式 S → AaB，且当前项目为 $S \to Aa \cdot B, b$。

- 假设 B 的产生式为 B → c。
- 新项目为 $B \to \cdot c, b$。
- 因为在·后面的β为空（即·后没有符号），直接继承展望符 b。

例子2：FIRST集传播

考虑产生式 S → AaB，且当前项目为 $S \to Aa \cdot Bc, b$。

- 假设 B 的产生式为 B → d。
- 新项目为 $B \to \cdot d, FIRST(c)$。
- 因为在·后面有符号 c，新项目的展望符是 `FIRST(c)`。

例子3：FIRST集和继承规则结合

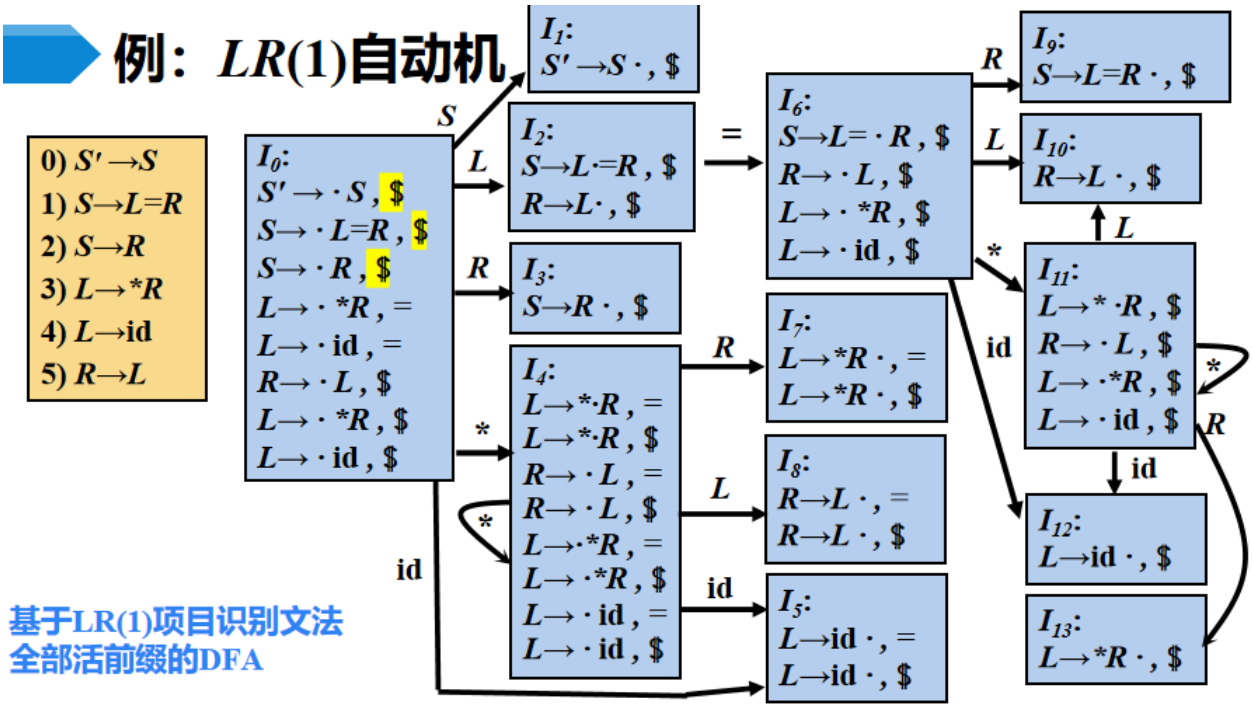
考虑产生式 S → AaB，且当前项目为 $S \to Aa \cdot B\beta, b$。

- 假设 B 的产生式为 B → d，且 β = f。
- 新项目为 $B \to \cdot d, FIRST(fb)$。
- 因为在·后面有符号 β，新项目的展望符是 `FIRST(fb)`。

LR(1)自动机

例：LR(1)自动机

0) $S' \rightarrow S$
1) $S \rightarrow L=R$
2) $S \rightarrow R$
3) $L \rightarrow *R$
4) $L \rightarrow id$
5) $R \rightarrow L$

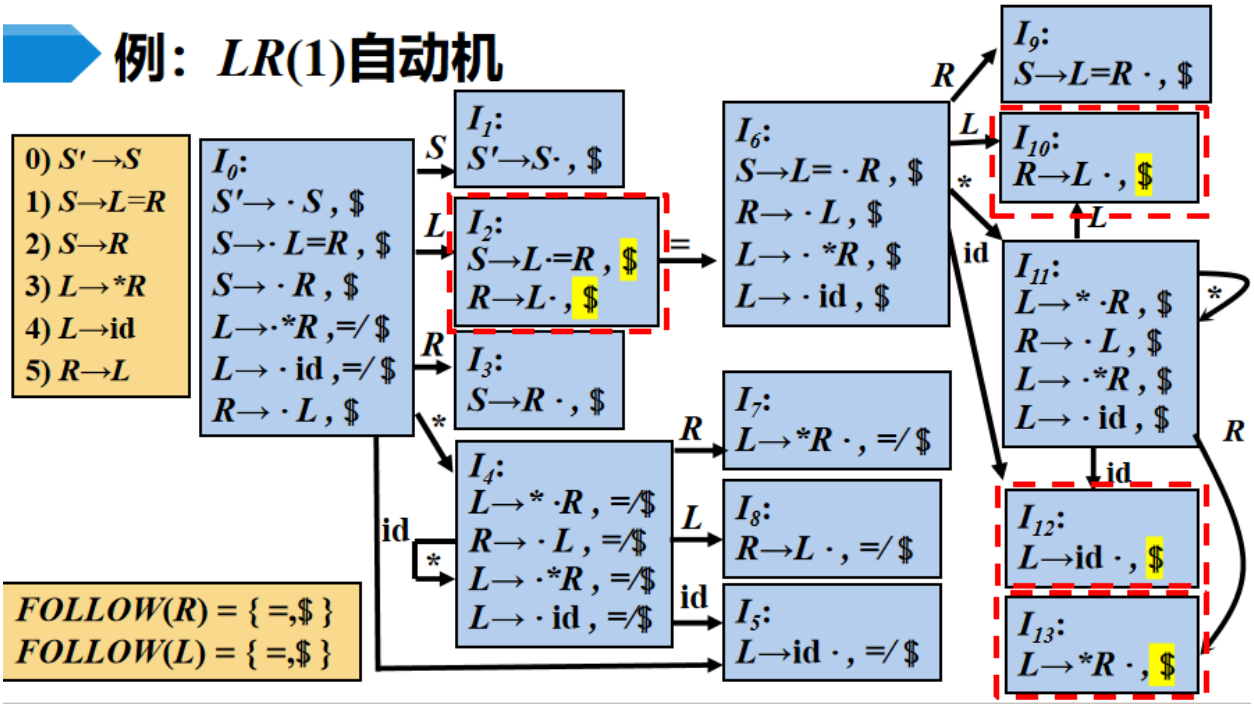


基于LR(1)项目识别文法
全部活前缀的DFA

赋值语句文法的 LR(1)分析表

文法
0) $S' \rightarrow S$
1) $S \rightarrow L=R$
2) $S \rightarrow R$
3) $L \rightarrow *R$
4) $L \rightarrow id$
5) $R \rightarrow L$

状态	ACTION				GOTO		
	*	id	=	$	S	L	R
0	s4	s5			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s4	s5				8	7
5			r4	r4			
6	s11	s12				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s11	s12				10	13
12				r4			
13				r3			



将SLR的一个项目根据展望符的不同进行分裂

如果**除展望符外**，两个LR(1)项目集是**相同的**，则称这两个LR(1)**项目集是**同心的

LR(1)项目集闭包

$$CLOSURE(I) = I \cup \{ [B \rightarrow \cdot\gamma, b] \mid [A \rightarrow \alpha \cdot B\beta, a] \in CLOSURE(I), B \rightarrow \gamma \in P, b \in FIRST(\beta a) \}$$

```
SetOfItems CLOSURE ( I ) {
    repeat
        for ( I中的每个项[A → α·Bβ， a ])
            for (G′的每个产生式B → γ)
                for ( FIRST (βa)中的每个符号b )
                    将[ B → ·γ ， b]加入到集合I中；
    until 不能向I中加入更多的项；
    until I ；
}
```

GOTO函数

$$GOTO(I, X) = CLOSURE(\{[A \rightarrow \alpha X \cdot \beta, a] \mid [A \rightarrow \alpha \cdot X\beta, a] \in I\})$$

```
SetOfItems GOTO (I, X) {
        将J初始化为空集;
        for (I中的每个项[A → α·Xβ, a])
                将项[A → αX·β, a]加入到集合J中;
        return CLOSURE (J);
}
```

为文法G' 构造LR(1)项集族

```
void items (G') {
        将C初始化为{CLOSURE ({[S' → ·S, $]})} ;
        repeat
                for (C中的每个项集I)
                        for (每个文法符号X)
                                if (GOTO(I, X)非空且不在C中)
                                        将GOTO(I, X)加入C中;
        until 不再有新的项集加入到C中;
}
```

LR(1)自动机的形式化定义

- 文法

$$G=(V_N, V_T, P, S)$$

- $LR(1)$自动机

$$M=(C, V_N \cup V_T, GOTO, I_0, F)$$

  - $C=\{I_0\} \cup \{I \mid \exists J \in C, X \in V_N \cup V_T, I=GOTO(J,X)\}$
  - $I_0=CLOSURE(\{S' \rightarrow \cdot S, \$\})$
  - $F=\{ CLOSURE(\{S' \rightarrow S \cdot, \$\}) \}$

- 构造G'的规范$LR(1)$项集族$C=\{I_0, I_1, \ldots, I_n\}$
- 根据I_i构造得到状态i。状态i的语法分析动作按照下面的方法决定：
  - *if* $[A \rightarrow \alpha \cdot a\beta, b] \in I_i$ *and* $GOTO(I_i, a)=I_j$ *then* $ACTION[i, a]=sj$
  - *if* $[A \rightarrow \alpha \cdot B\beta, b] \in I_i$ *and* $GOTO(I_i, B)=I_j$ *then* $GOTO[i, B]=j$
  - *if* $[A \rightarrow \alpha \cdot, a] \in I_i$且$A \neq S'$ *then* $ACTION[i, a]=rj$
  - *if* $[S' \rightarrow S \cdot, \$] \in I_i$ *then* $ACTION[i, \$]=acc$;
- 没有定义的所有条目都设置为“*error*”

如果$LR(1)$分析表中没有语法分析动作冲突，那么给定的文法就称为$LR(1)$文法

各种LR分析表构造方法的不同之处在于归约项目的处理上

- if $[A\to\alpha\cdot, c]\in I_i$

$A=S'$ $\quad ACTION[i,\$]=acc$

$A\neq S'$

- LR(0) $\quad for\ \forall a\in V_T\cup\{\$\} \quad do\ ACTION[i,a]=r_j$（j是产生式$A\to\alpha$的编号）
- SLR(1) $\quad for\ \forall a\in FOLLOW(A) \quad do\ ACTION[i,a]=r_j$
- LR(1) $\quad$ 精准归约 $\quad ACTION[i,c]=r_j$

LR(1)分析实际上是根据后继符集合的不同，
将原始的LR(0)状态分裂成不同的LR(1)状态

> 注：LR（1）文法都是无二义性但是可能是左递归

4.4.4 LALR分析

LALR (lookahead-LR)分析的基本思想

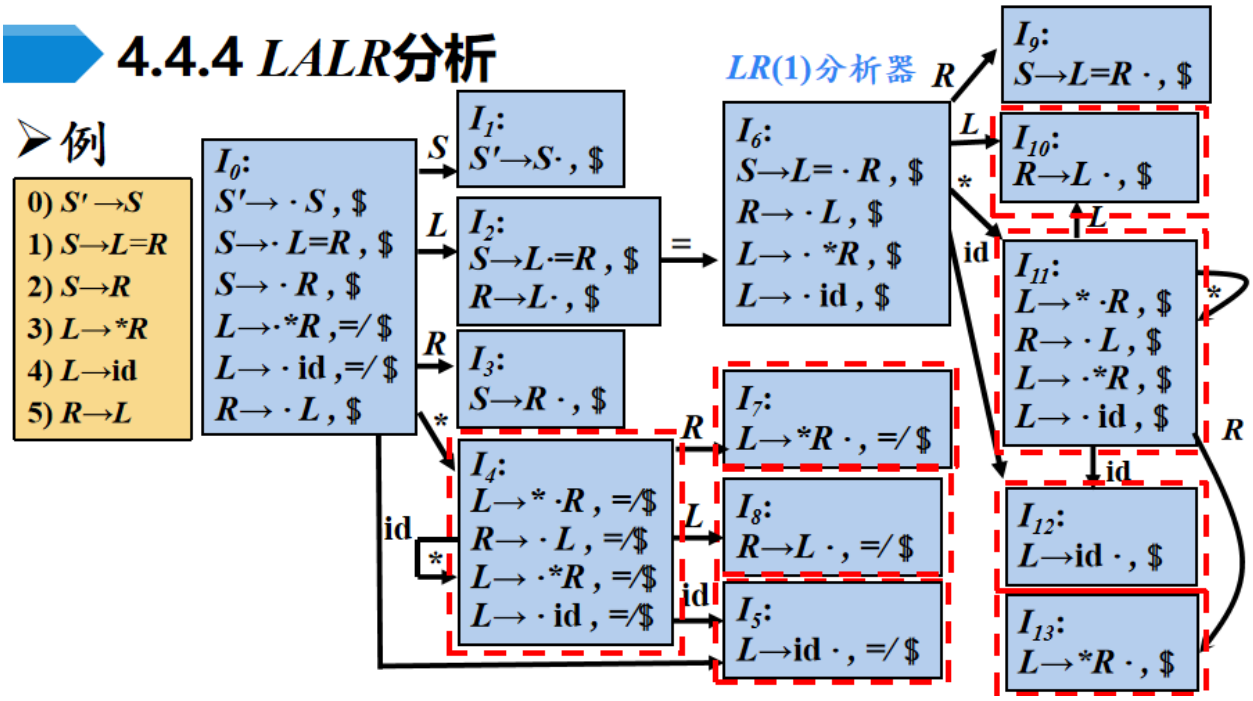
如果**除展望符外**，两个LR(1)项目集是**相同的**，则称这两个LR(1)**项目集是**同心的

- 寻找具有相同核心的*LR* (1) 项集，并将这些项集合并为一个项集。所谓项集的核心就是其第一分量的集合
- 然后根据合并后得到的项集族构造语法分析表
- 如果分析表中没有语法分析动作冲突，给定的文法就称为*LALR* (1) 文法，就可以根据该分析表进行语法分析

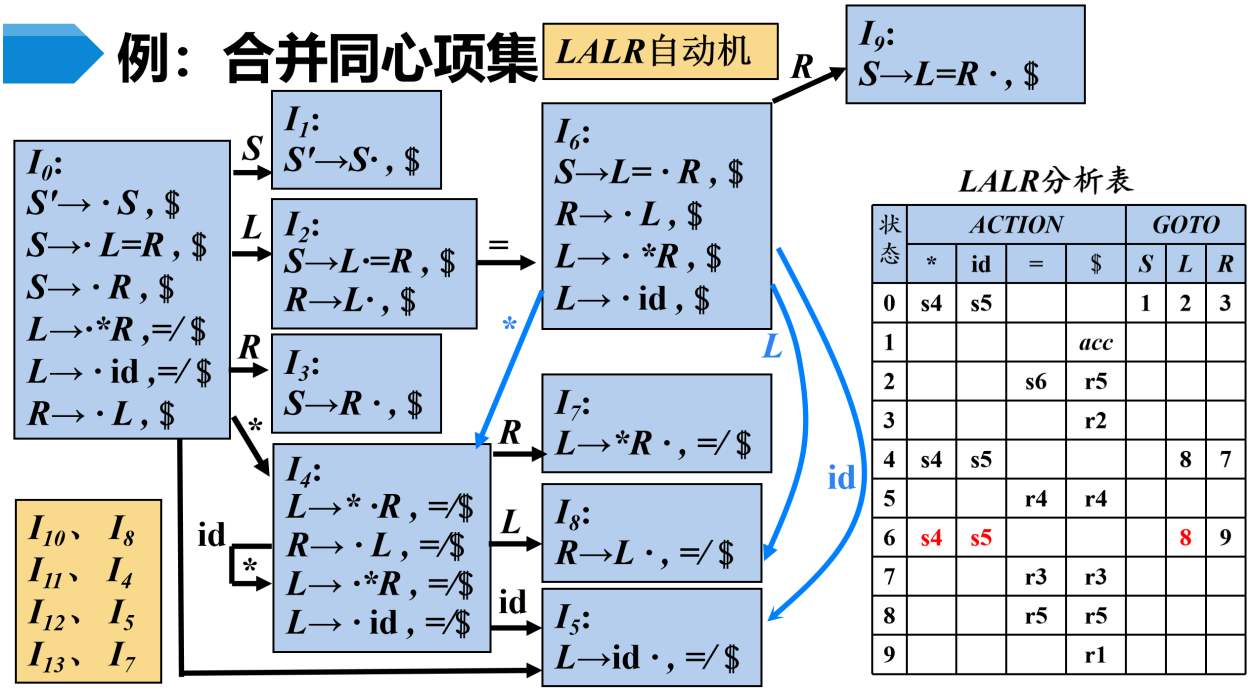
4.4.4 *LALR*分析

- 例

0) $S' \rightarrow S$
1) $S \rightarrow L=R$
2) $S \rightarrow R$
3) $L \rightarrow *R$
4) $L \rightarrow id$
5) $R \rightarrow L$



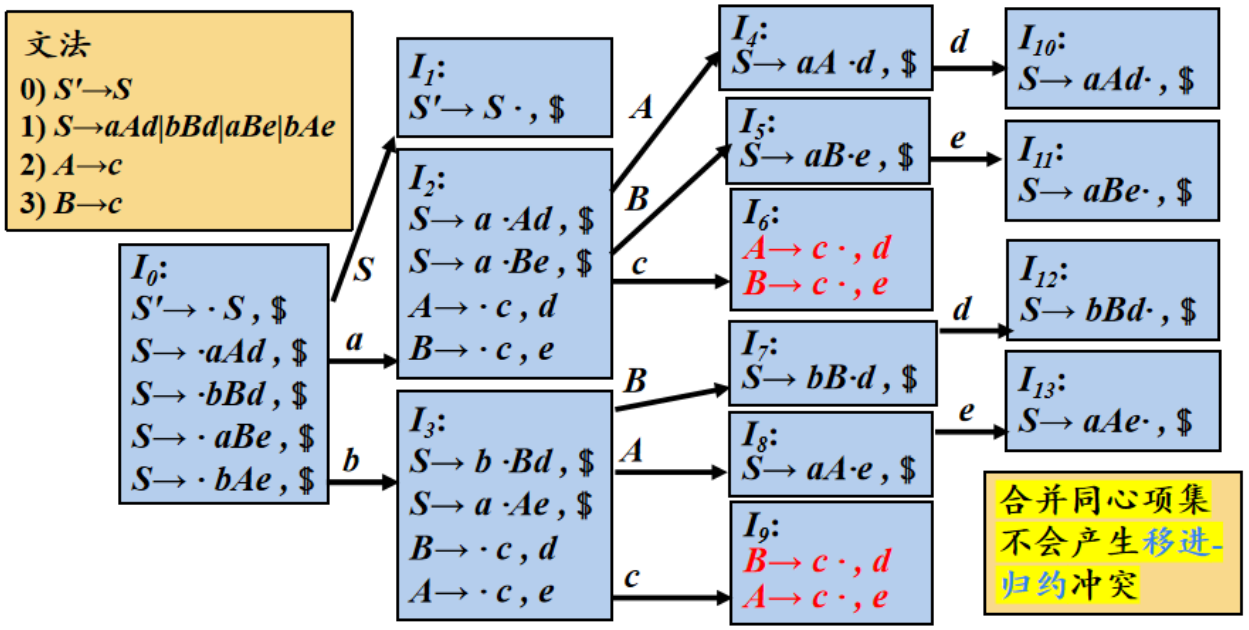
例：合并同心项集

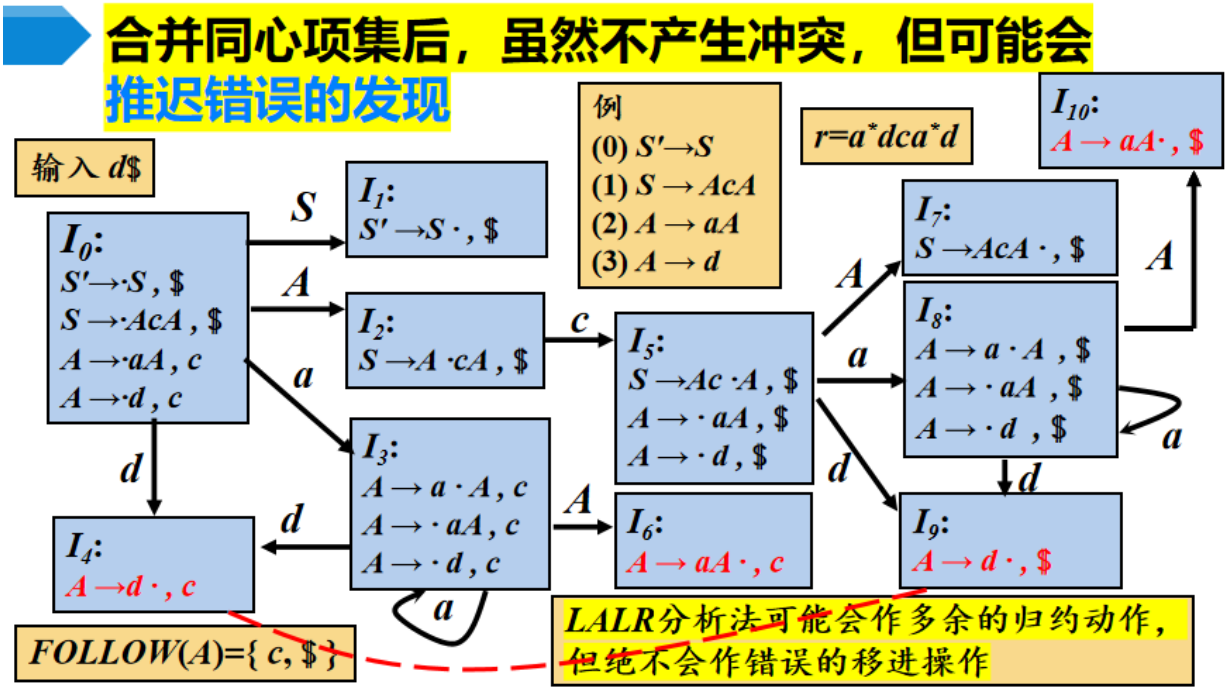


LALR分析表

状态	ACTION				GOTO		
	*	id	=	$	S	L	R
0	s4	s5			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s4	s5				8	7
5			r4	r4			
6	s4	s5				8	9
7			r3	r3			
8			r5	r5			
9				r1			

合并同心项集时产生归约-归约冲突，但是不会产生移入-归约冲突





LALR(1)的特点

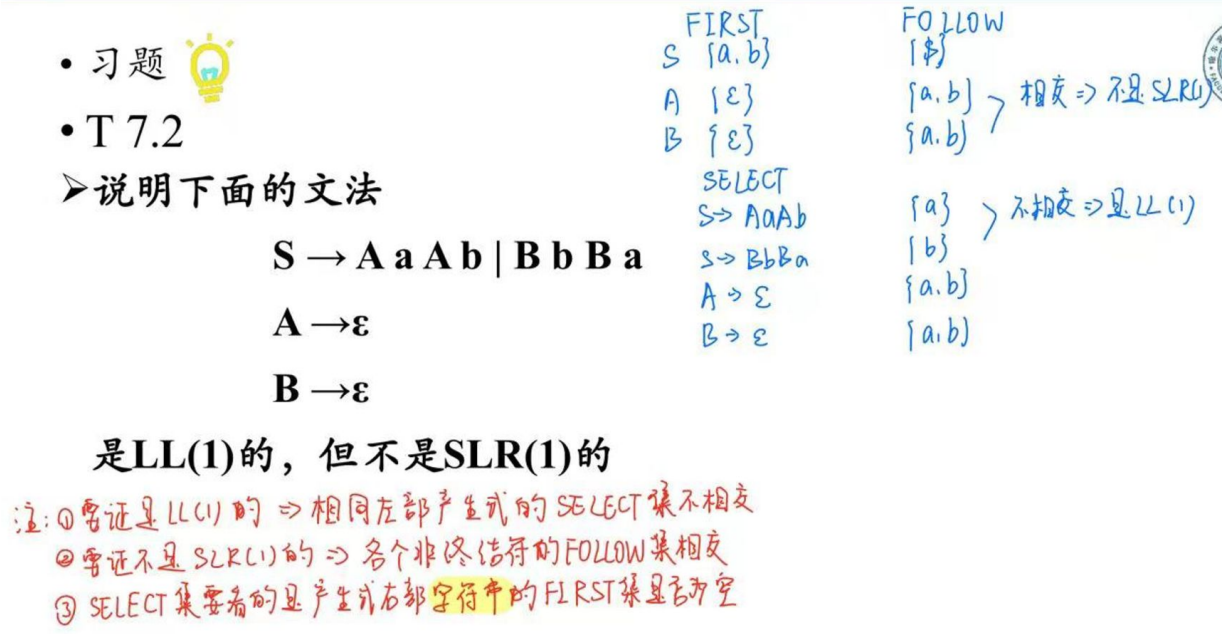
- 形式上与LR(1)相同
- 大小上与LR(0)/SLR相当
- 分析能力介于SLR和LR(1)二者之间

$$LR(0) < SLR < LALR(1) < LR(1)$$

- 合并后的展望符集合仍为FOLLOW集的子集

> 注：一个LR（1）文法合并同心项集后若不是LALR（1）文法，则可能存在归约/归约冲突

习题

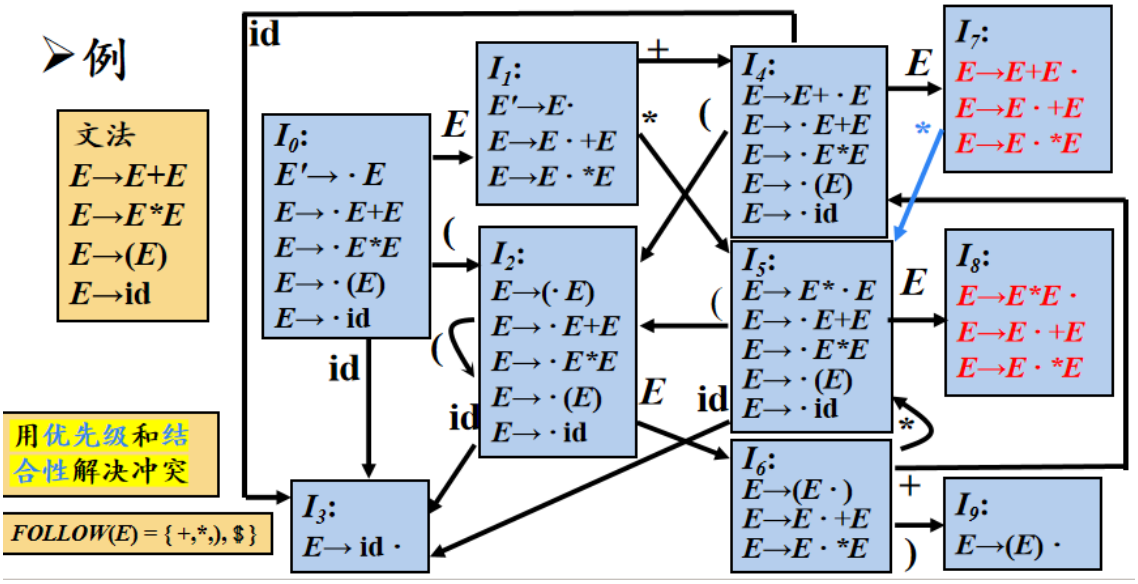


注：判断是否是SLR（1）的主要还是看SLR分析表中是否有冲突，构造SLR自动机

4.4.5 二义性文法的LR分析

- 每个二义性文法**都不是LR的**
  - 但是二义性文法可以用LR分析

二义性算术表达式文法的LR(0)分析器



状态	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

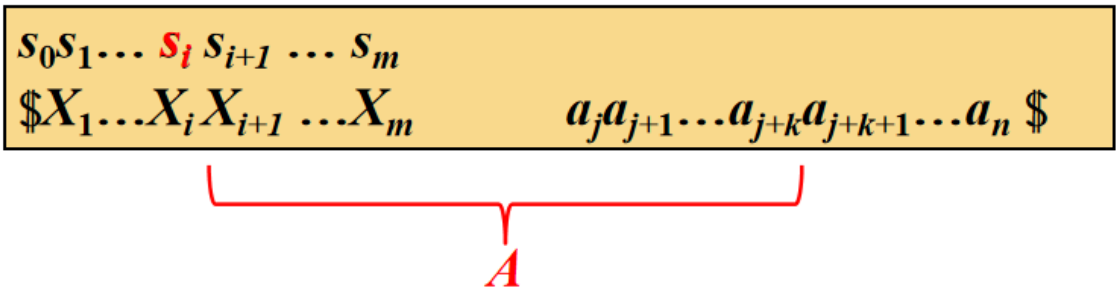
- 某些类型的二义性文法在语言的描述和实现中很有用
  - 更简短、更自然

- 二义性文法的使用
  - 应该**保守地使用二义性文法**，并且必须在严格控制之下使用，因为稍有不慎就会导致语法分析器所识别的语言出现偏差

4.4.6 LR分析中的错误处理

- 语法错误的检测
  - 当LR分析器在查询语法分析**动作表**并**发现一个报错条目**时，就检测到了一个语法错误
- **错误恢复策略**
  - 恐慌模式错误恢复
  - 短语层次错误恢复

恐慌模式错误恢复



- 从栈顶向下扫描，直到发现某个状态s_i，它有一个对应于某个非终结符A的$GOTO$目标，可以认为从这个A推导出的串中包含错误
- 然后丢弃0个或多个输入符号，直到发现一个可能合法地紧跟在A之后的符号a为止
- 之后将$s_{i+1} = GOTO(s_i, A)$压入栈中，继续进行正常的语法分析
- 实践中可能会选择多个这样的非终结符A。通常这些非终结符代表了主要的程序段，比如表达式、语句或块

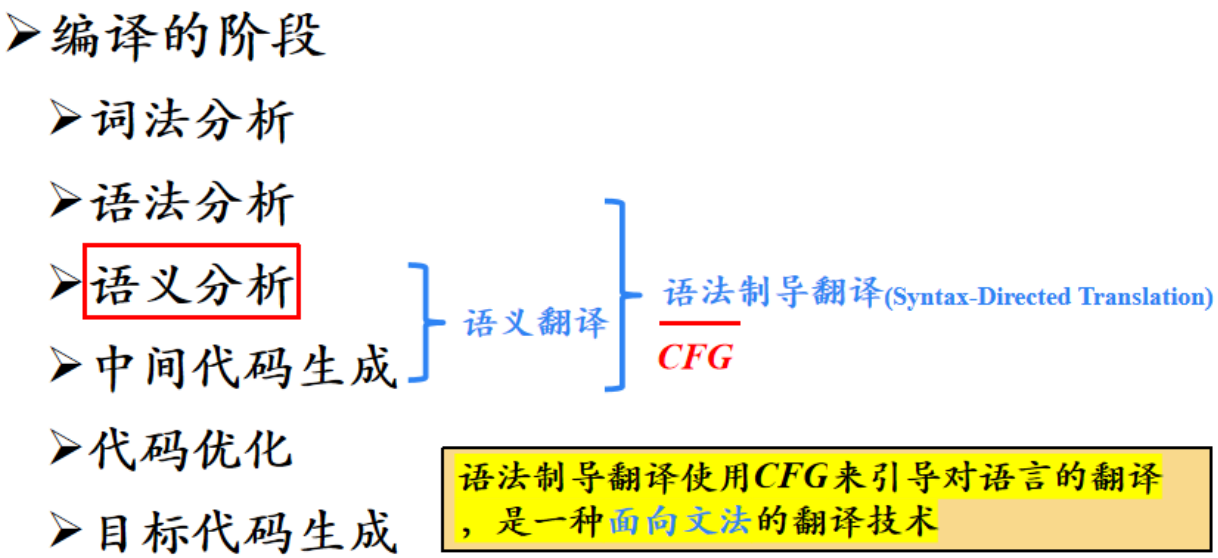
- 从栈顶向下扫描→状态往回退

短语层次错误恢复

- 检查LR分析表中的每一个报错条目，并根据**语言的使用方法**来决定程序员所犯的何种错误最有可能引起这个语法错误
- 然后构造出适当的恢复过程

习题

第五章 语法制导翻译



- **语法制导翻译的基本思想**
  - 如何**表示语义信息**？
    - 为CFG中的文法符号**设置语义属性**，用来表示语法成分对应的语义信息
  - 如何**计算语义属性**？
    - 文法符号的**语义属性值**是用与文法符号所在**产生式（语法规则）**相关联的**语义规则**来**计算**的
    - 对于给定的输入串x，构建x的语法分析树，并利用与**产生式（语法规则）**相关联的**语义规则**来**计算**分析树中各结点对应的**语义属性值**

- 将**语义规则**同**语法规则（产生式）**联系起来要涉及两个概念
  - **语法制导定义（SDD）**
    - SDD是对CFG的推广
      1. 将每个**文法符号**和一个**语义属性集合**相关联
      2. 将每个**产生式**和一组**语义规则**相关联，这些规则用于**计算**该产生式中各文法符号的**属性值**

产生式	语义规则
$D \rightarrow T\ L$	$L.inh = T.type$
$T \rightarrow \text{int}$	$T.type = int$
$T \rightarrow \text{real}$	$T.type = real$
$L \rightarrow L_1, \text{id}$	$L_1.inh = L.inh$
…	…

- 如果X是一个文法符号，a是X的一个属性，则用X.a表示属性a在某个标号为X的分析树结点上的值

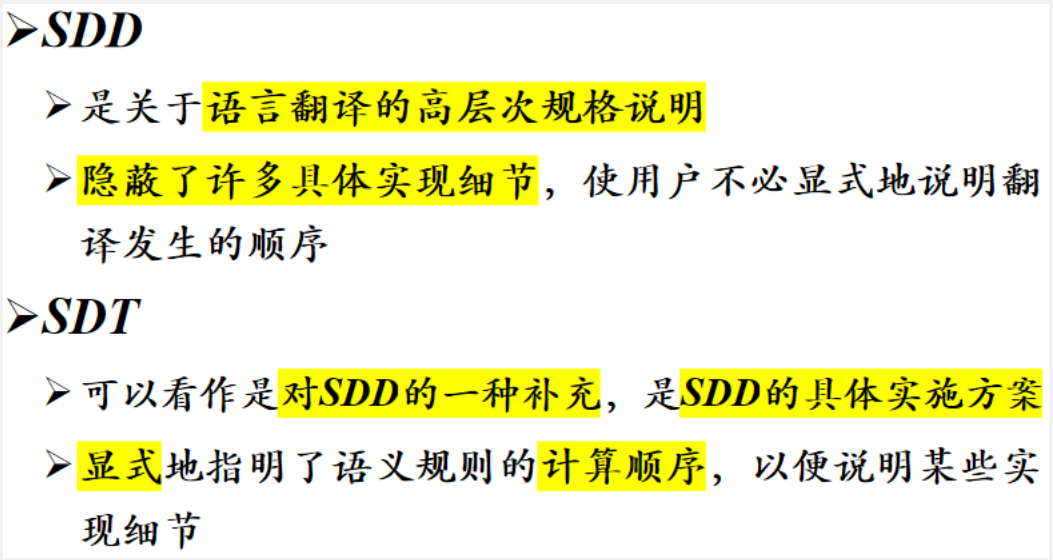
- **语法制导翻译方案 （SDT）**
  - SDT是在**产生式右部**嵌入了**程序片段**的CFG，这些程序片段称为**语义动作**。按照惯例，**语义动作放在花括号内**

➢例

$D \rightarrow T\ \{ L.inh = T.type \}\ L$
$T \rightarrow \text{int}\ \{ T.type = int \}$
$T \rightarrow \text{real}\ \{ T.type = real \}$
$L \rightarrow \{ L_1.inh = L.inh \}\ L_1, \text{id}$
…

一个语义动作在产生式中的位置决定了这个动作的执行时间

SDD VS SDT



5.1 语法制导定义SDD

- **文法符号的属性**
  - **综合属性** (synthesized attribute)
    - 在分析树结点 N上的**非终结符**A的**综合属性**只能通过 *N的子结点* 或 *N本身* 的属性值来定义
    - **终结符**可以具有**综合属性**

      终结符的综合属性值是由**词法分析器提供的词法值**，因此在SDD中没有计算终结符属性值的语义规则

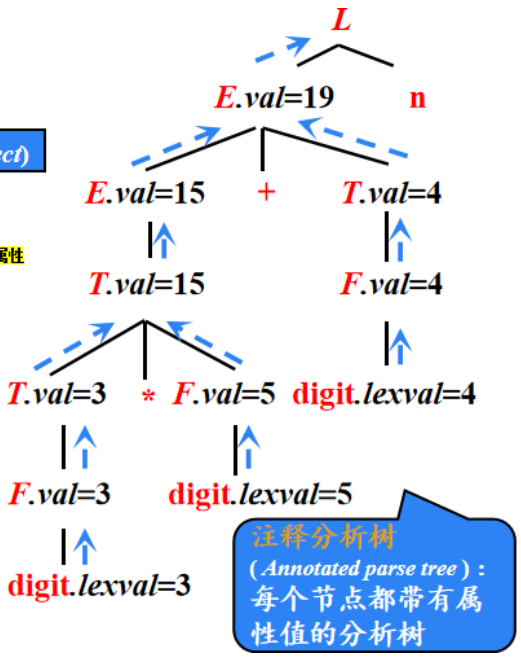
SDD：

副作用(Side effect)

产生式	语义规则
(1) $L \to E$ n	print($E.val$) 定义了一个虚的综合属性
(2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
(3) $E \to T$	$E.val = T.val$
(4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
(5) $T \to F$	$T.val = F.val$
(6) $F \to (E)$	$F.val = E.val$
(7) $F \to$ digit	$F.val =$ digit.$lexval$

输入：

3*5+4n

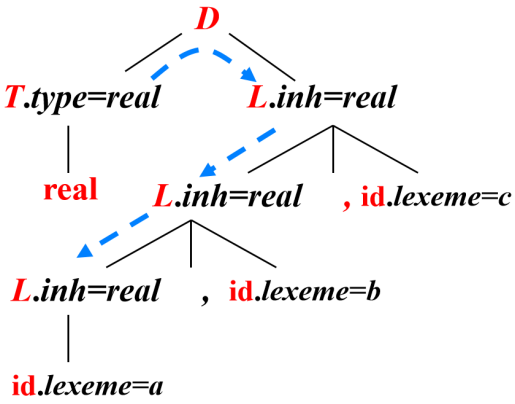


- **继承属性**（inherited attribute）
  - 在分析树结点 N上的**非终结符**A的**继承属性**只能通过 *N的父结点*、*N的兄弟结点*或 *N本身* 的属性值来定义
  - **终结符没有继承属性**。

    终结符从词法分析器处获得的属性值被归为综合属性值

SDD：

	产生式	语义规则
(1)	$D \to T\ L$	$L.inh = T.type$
(2)	$T \to$ int	$T.type = int$
(3)	$T \to$ real	$T.type = real$
(4)	$L \to L_1$, id	$L_1.inh = L.inh$
		addtype(id.$lexeme$, $L.inh$)
(5)	$L \to$ id	addtype(id.$lexeme$, $L.inh$)



输入：

real a , b , c

- **属性文法**
  - 一个**没有副作用的SDD**有时也称为属性文法

    - 属性文法的规则**仅仅通过其它属性值和常量来定义一个属性值**

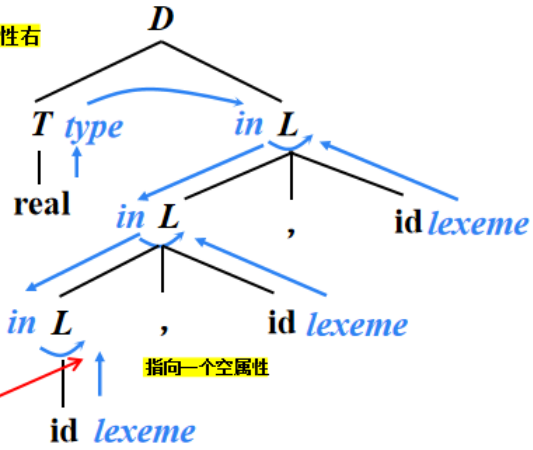
- **SDD的求值顺序**
    - 语义规则建立了**属性之间的依赖关系**，在对语法分析树节点的一个属性求值之前，必须**首先求出**这个属性值**所依赖的所有属性值**

- **依赖图**
    - 依赖图是一个描述了分析树中结点属性间**依赖关系**的**有向图**
    - 分析树中每个标号为X的结点的每个属性a都对应着依赖图中的一个结点
    - 如果**属性X.a的值依赖于属性Y.b的值**，则依赖图中**有一条从Y.b的结点指向X.a的结点的有向边**

SDD：

继承属性左，综合属性右

	产生式	语义规则
(1)	$D \rightarrow T\ L$	$L.in = T.type$
(2)	$T \rightarrow$ int	$T.type = int$
(3)	$T \rightarrow$ real	$T.type = real$
(4)	$L \rightarrow L_1$, id	$L_1.in = L.in$
		$addtype(\text{id}.lexeme, L.in)$
(5)	$L \rightarrow$ id	$addtype(\text{id}.lexeme, L.in)$



输入：

real *a* , *b* , *c*

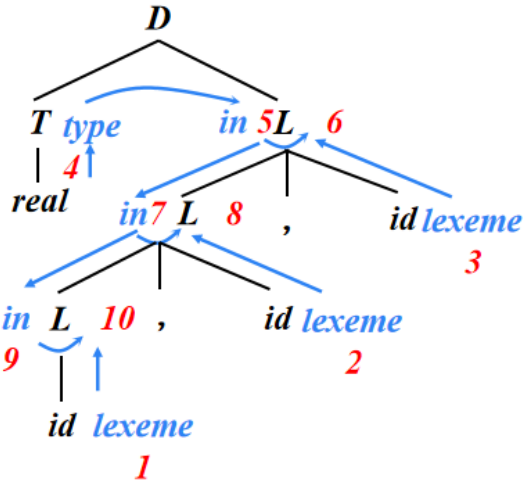
- **属性值的计算顺序**
    - 可行的求值顺序是满足下列条件的结点序列N1, N2, ... , Nk ：**如果依赖图中有一条从结点Ni到 Nj 的边(Ni→Nj)，那么i < j**（即：在节点序列中，Ni 排在Nj 前面）
    - 这样的排序将一个有向图变成了一个线性排序，这个排序称为这个图的**拓扑排序**
    - 从计算的角度看，给定一个SDD，**很难确定**是否存在某棵语法分析树，使得SDD的属性之间**存在循环依赖**关系

SDD：

	产生式	语义规则
(1)	$D \rightarrow T\ L$	$L.in = T.type$
(2)	$T \rightarrow$ int	$T.type = int$
(3)	$T \rightarrow$ real	$T.type = real$
(4)	$L \rightarrow L_1$, id	$L_1.in = L.in$ $addtype(\text{id}.lexeme, L.in)$
(5)	$L \rightarrow$ id	$addtype(\text{id}.lexeme, L.in)$

输入：

real *a* , *b* , *c*



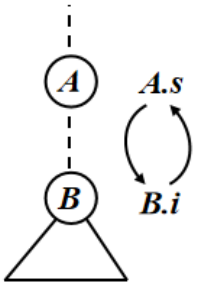
拓扑排序：

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

4, 3, 2, 1, 5, 7, 6, 9, 8, 10

- 对于只具有综合属性的SDD，可以按照任何自底向上的顺序计算它们的值
- 对于同时具有继承属性和综合属性的SDD，不能保证存在一个顺序来对各个节点上的属性进行求值
- 例

产生式	语义规则
$A \rightarrow B$	$A.s = B.i$ $B.i = A.s + 1$



如果图中没有环，那么至少存在一个拓扑排序

5.2 S-属性定义与L-属性定义

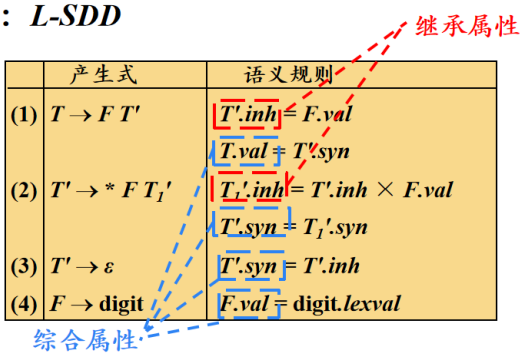
- 存在一个SDD的有用子类，它们能够**保证**对每棵语法分析树都**存在一个求值顺序**，因为它们**不允许产生带有环的依赖图**

- 不仅如此，接下来介绍的两类SDD可以和**自顶向下**及**自底向上**的语法分析过程一起高效地实现
    - S-属性定义 (S-Attributed Definitions, **S-SDD**)
    - L-属性定义 (L-Attributed Definitions, **L-SDD**)

- **S-属性定义（S-SDD）**
    - **仅仅使用综合属性**的SDD称为S属性的SDD，或S-属性定义、S-SDD
    - 如果一个SDD是S属性的，可以按照语法分析树节点的任何**自底向上**顺序来**计算**它的各个属性值
    - S-属性定义可以在**自底向上**的语法分析过程中实现
- **L-属性定义（L-SDD）**
    - 直观含义：在一个产生式所关联的各属性之间，依赖图的边**可以从左到右**，但**不能从右到左**(因此称为L属性的，L是Left的首字母)
    - 正式定义：
        - 一个SDD是L-属性定义，当且仅当它的每个属性**要么是一个综合属性，要么是满足如下条件的继承属性**：假设存在一个产生式A→X1X2...Xn，其右部符号Xi (1≤ i ≤ n)的继承属性仅依赖于下列属性：
            1. A的继承属性（不能是综合属性）
            2. 产生式中Xi左边的符号 X1, X2, ... , Xi-1 的属性
            3. Xi本身的属性，但**Xi 的全部属性不能在依赖图中形成环路**

> 💡 **每个S-属性定义都是L-属性定义**

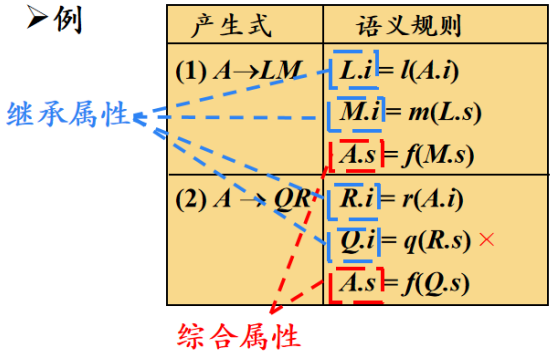
例：*L-SDD*



	产生式	语义规则
(1)	$T \to F\ T'$	$T'.inh = F.val$
		$T.val = T'.syn$
(2)	$T' \to *\ F\ T_1'$	$T_1'.inh = T'.inh \times F.val$
		$T'.syn = T_1'.syn$
(3)	$T' \to \varepsilon$	$T'.syn = T'.inh$
(4)	$F \to$ digit	$F.val =$ digit.*lexval*

非*L*属性的*SDD*

➢例

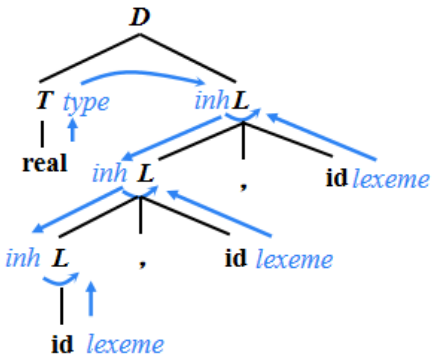


产生式	语义规则
(1) $A \to LM$	$L.i = l(A.i)$
	$M.i = m(L.s)$
	$A.s = f(M.s)$
(2) $A \to QR$	$R.i = r(A.i)$
	$Q.i = q(R.s) \times$
	$A.s = f(Q.s)$

- **虚属性**

SDD：

	产生式	语义规则
(1)	$D \to T\ L$	$L.inh = T.\ type$
(2)	$T \to$ int	$T.type = int$
(3)	$T \to$ real	$T.type = real$
(4)	$L \to L_1$, id	$L_1.inh = L.inh$
		addtype(id.*lexeme*, *L.inh*)
(5)	$L \to$ id	*addtype*(id.*lexeme*, *L.inh*)



输入：

real *a* , *b* , *c*

5.3 语法制导翻译方案SDT

- **SDT可以看作是SDD的具体实施方案**

➢ ***SDD***定义了各属性的计算方法（计算规则） 怎么算?

➢ ***SDT***进一步明确了各属性的计算时机（计算顺序） 怎么算? + 何时算?

无循环依赖SDD的SDT

SDD

	产生式	语义规则
(1)	$T \to F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
(2)	$T' \to *\ F\ T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
(3)	$T' \to \varepsilon$	$T'.syn = T'.inh$
(4)	$F \to \text{digit}$	$F.val = \text{digit}.lexval$

SDT

1) $T \to F$ { $T'.inh = F.val$ } T' { $T.val = T'.syn$ }
2) $T' \to *F$ { $T_1'.inh = T'.inh \times F.val$ } T_1' { $T'.syn = T_1'.syn$ }
3) $T' \to \varepsilon$ { $T'.syn = T'.inh$ }
4) $F \to \text{digit}$ { $F.val = \text{digit}.lexval$ }

- 两类重要SDD的SDT实现（这两种SDT可以在语法分析中实现）
  - 基本文法可以使用**LR**分析技术，且**SDD是S属性的**
  - 基本文法可以使用**LL**分析技术，且**SDD是L属性的**

将S-SDD转换为SDT

- 将一个S-SDD转换为SDT的方法：**将每个语义动作都放在产生式的最后**

S-SDD

产生式	语义规则
(1) $L \to E\ \text{n}$	$L.val = E.val$
(2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
(3) $E \to T$	$E.val = T.val$
(4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
(5) $T \to F$	$T.val = F.val$
(6) $F \to (E)$	$F.val = E.val$
(7) $F \to \text{digit}$	$F.val = \text{digit}.lexval$

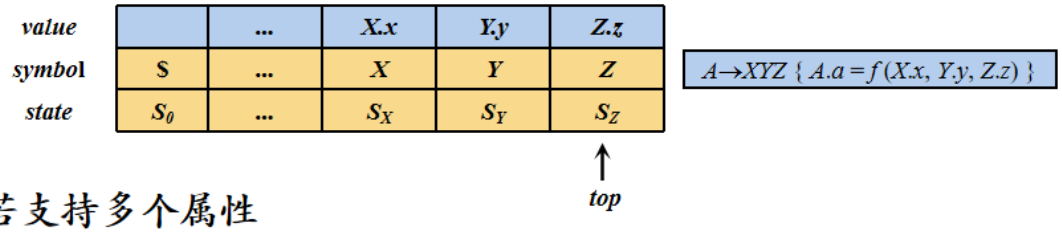
SDT

(1) $L \to E\ \text{n}$ { $L.val = E.val$ }
(2) $E \to E_1 + T$ { $E.val = E_1.val + T.val$ }
(3) $E \to T$ { $E.val = T.val$ }
(4) $T \to T_1 * F$ { $T.val = T_1.val \times F.val$ }
(5) $T \to F$ { $T.val = F.val$ }
(6) $F \to (E)$ { $F.val = E.val$ }
(7) $F \to \text{digit}$ { $F.val = \text{digit}.lexval$ }

- 如果一个**S-SDD的基本文法**可以使用**LR分析技术**，那么它的**SDT**可以在**LR语法分析**过程中实现

- 当**归约发生**时**执行相应的语义动作**

- **对语法分析器进行扩展**
  - 为每个栈记录**增加属性值字段**，存放文法符号的**综合属性值**
  - 在**每次归约**时调用**计算综合属性值**的语义子程序

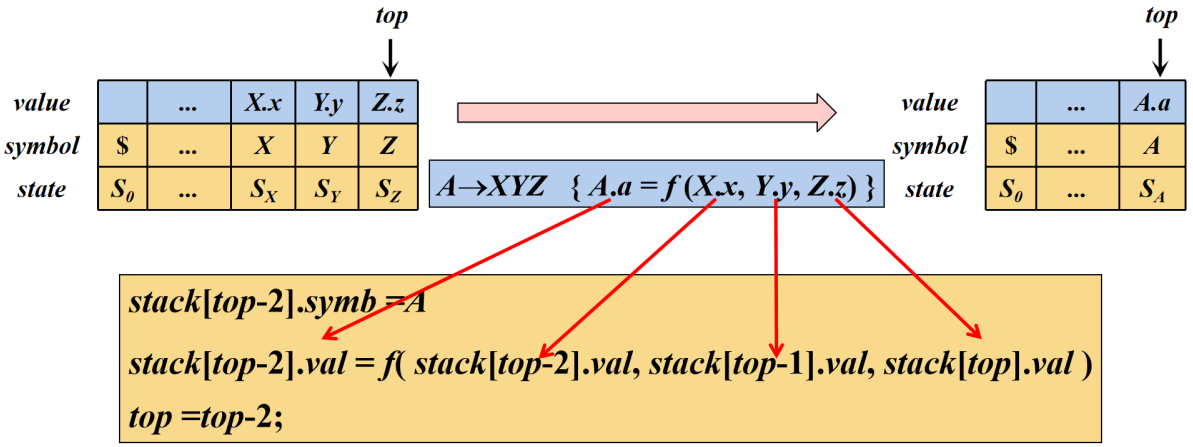


➢若支持多个属性

➢方法1：使栈记录变得足够大

➢方法2：在栈记录中存放指针

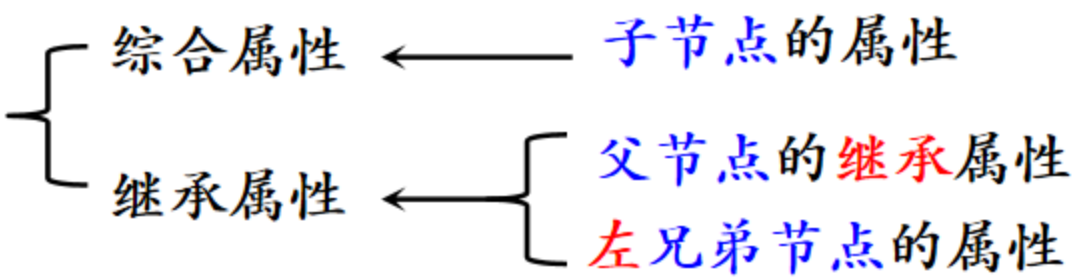
- **将语义动作中的抽象定义式改写成具体可执行的栈操作**



产生式	语义动作	
(1)$E' \rightarrow E$	print(*E.val*)	{ print (*stack*[*top*].*val*);}
(2)$E \rightarrow E_1 + T$	*E.val* = E_1.*val* + *T.val*	{ *stack*[*top*-2].*val* = *stack*[*top*-2].*val* + *stack*[*top*].*val* ; *top*=*top*-2; } 运算符也在栈里
(3)$E \rightarrow T$	*E.val* = *T.val*	
(4)$T \rightarrow T_1 * F$	*T.val* = T_1.*val* × *F.val*	{ *stack*[*top*-2].*val* = *stack*[*top*-2].*val* × *stack*[*top*].*val* ; *top*=*top*-2; }
(5)$T \rightarrow F$	*T.val* = *F.val*	
(6)$F \rightarrow (E)$	*F.val* = *E.val*	{ *stack*[*top*-2].*val* = *stack*[*top*-1].*val*; *top*=*top*-2; } 括号也在栈里
(7)$F \rightarrow$ digit	*F.val* = digit.*lexval*	

- 遇到可以规约的式子进行规约回退并删除状态栏和符号栏和更新属性
- 加入新规约好的符号
- 利用上一栏的状态
- 遇到新加入的规约好的符号
- 生成新的状态并加入

将L-SDD转换为SDT



- **规则：**
  - 将计算一个产生式**左部**符号的**综合属性**的动作放置在这个**产生式右部的最右端**
  - 将计算某个非终结符号A的**继承属性**的动作插入到产生式右部中**紧靠在A的本次出现之前**的位置上

➢ *L-SDD*

	产生式	语义规则
(1)	$T \to F\ T'$	$T'.inh = F.val$
		$T.val = T'.syn$
(2)	$T' \to *\ F\ T_1'$	$T_1'.inh = T'.inh \times F.val$
		$T'.syn = T_1'.syn$
(3)	$T' \to \varepsilon$	$T'.syn = T'.inh$
(4)	$F \to \text{digit}$	$F.val = \text{digit}.lexval$

➢ *SDT*

1) $T \to F\ \{\ T'.inh = F.val\ \}\ T'\ \{\ T.val = T'.syn\ \}$
2) $T' \to *F\ \{\ T_1'.inh = T'.inh \times F.val\ \}\ T_1'\ \{\ T'.syn = T_1'.syn\ \}$
3) $T' \to \varepsilon\ \{\ T'.syn = T'.inh\ \}$
4) $F \to \text{digit}\ \{\ F.val = \text{digit}.lexval\ \}$

- 如果一个**L-SDD的基本文法**可以使用**LL分析技术**，那么它的**SDT**可以在**LL或LR语法分析**过程中实现
  - 在非递归的预测分析过程中进行语义翻译
  - 在递归的预测分析过程中进行语义翻译
  - 在LR分析过程中进行语义翻译

5.4 L-属性定义的自顶向下翻译

- **LL（1）文法 + L-SDD（自顶向下）**

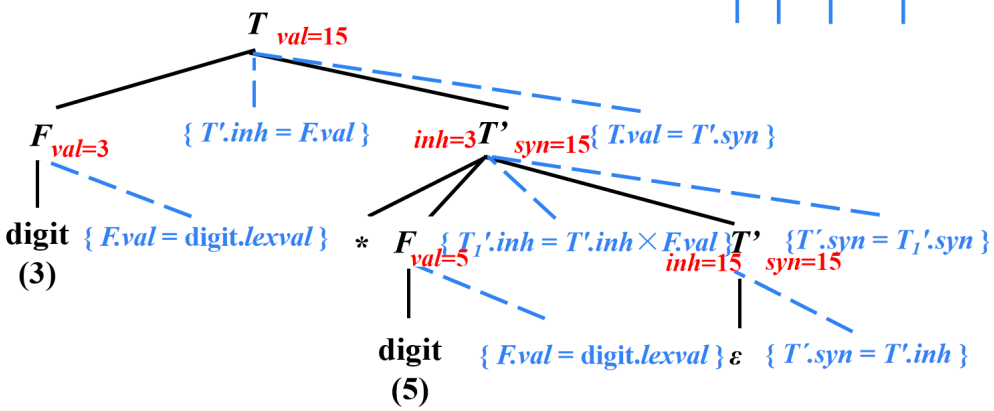
例：

1) $T \rightarrow F$ { $T'.inh = F.val$ } T' { $T.val = T'.syn$ }
2) $T' \rightarrow *F$ { $T_1'.inh = T'.inh \times F.val$ } T_1' { $T'.syn = T_1'.syn$ }
3) $T' \rightarrow \varepsilon$ { $T'.syn = T'.inh$ }
4) $F \rightarrow$ digit { $F.val$ = digit.$lexval$ }

输入：

3 * 5

digit * digit

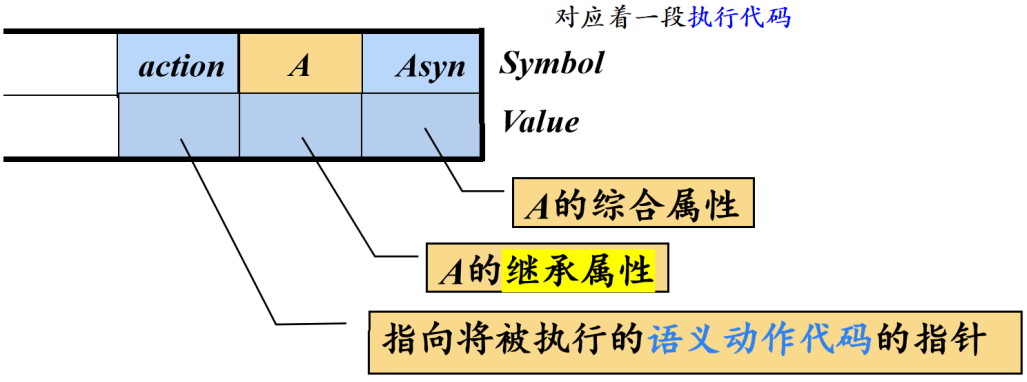


- 在预测分析的同时实现语义翻译
  - 在**非递归的预测分析**过程中进行翻译（显式地维护一个**栈结构**来模拟**最左推导**过程）
  - 在**递归的预测分析**过程中进行翻译（基于**预测分析表**对**递归下降分析法**进行扩展）

5.4.1 在非递归的预测分析过程中进行翻译

➢扩展语法分析栈

（1）增加属性值（value）字段
（2）对于非终结符A，将继承属性和综合属性存放在不同的记录中
（3）终结符的综合属性存放在其记录的属性值字段
（4）增加动作记录用来存放指向语义动作代码的指针
（5）不光是动作记录，其实分析栈中的每一个记录都对应着一段执行代码



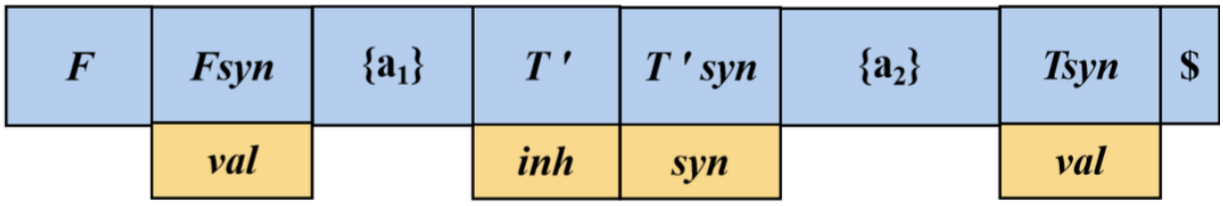
SDT

1) $T \rightarrow F \{a_1\}\ T' \{a_2\}$
2) $T' \rightarrow *F \{a_3\}\ T_1' \{a_4\}$
3) $T' \rightarrow \varepsilon \{a_5\}$
4) $F \rightarrow \text{digit} \{a_6\}$

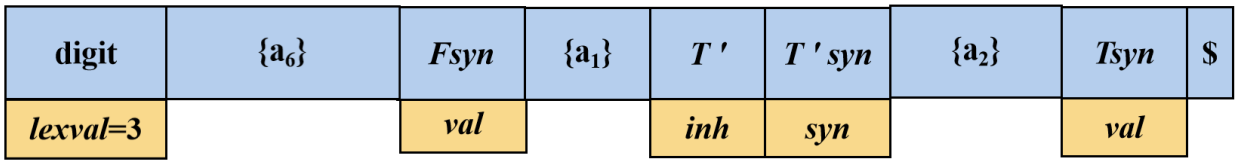
a_1 : $T'.inh = F.val$
a_2 : $T.val = T'.syn$
a_3 : $T_1'.inh = T'.inh \times F.val$
a_4 : $T'.syn = T_1'.syn$
a_5 : $T'.syn = T'.inh$
a_6 : $F.val = \text{digit}.lexval$

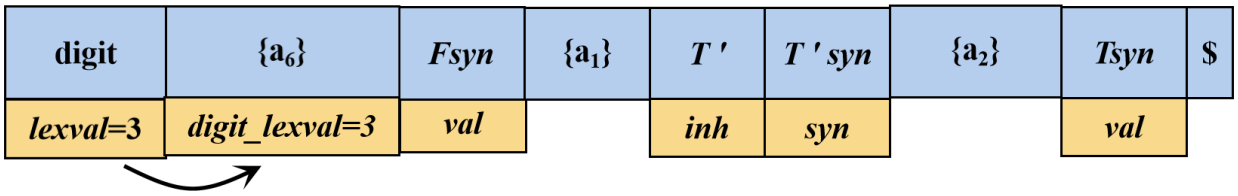
输入: 3 * 5

	继承属性	综合属性
T		val
T'	inh	syn
F		val

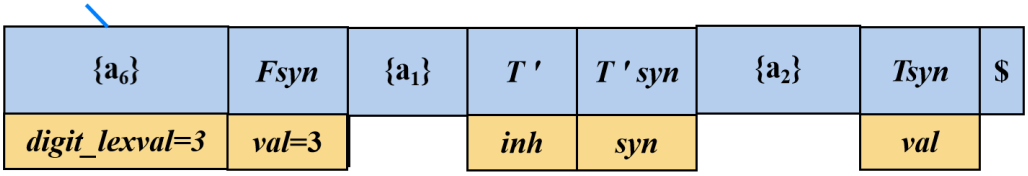


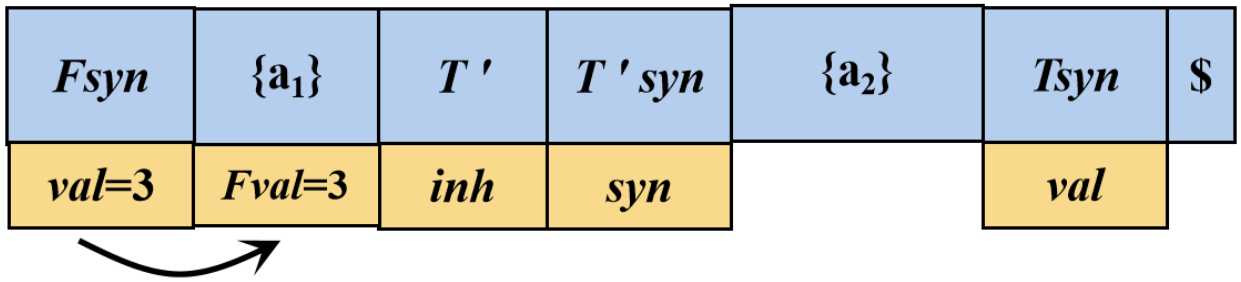
P62

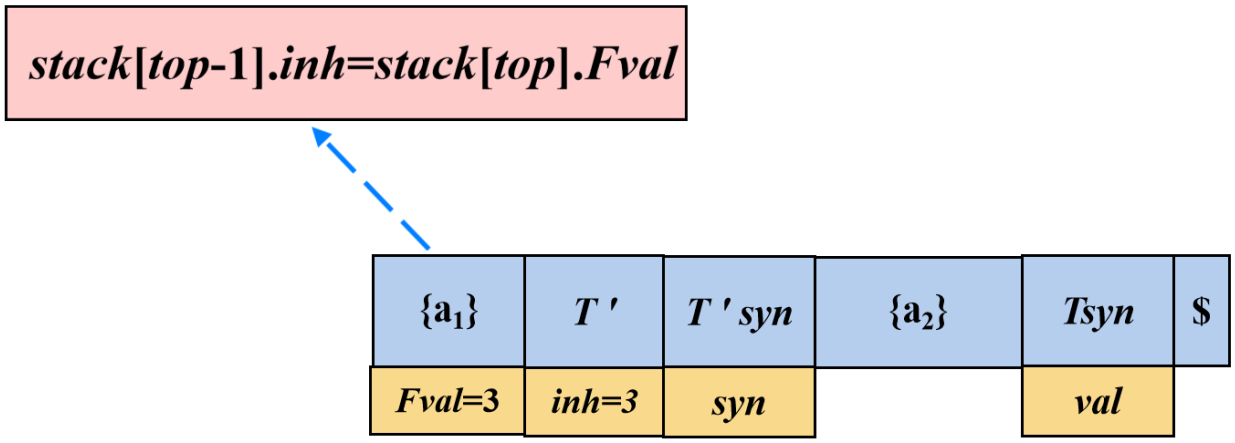


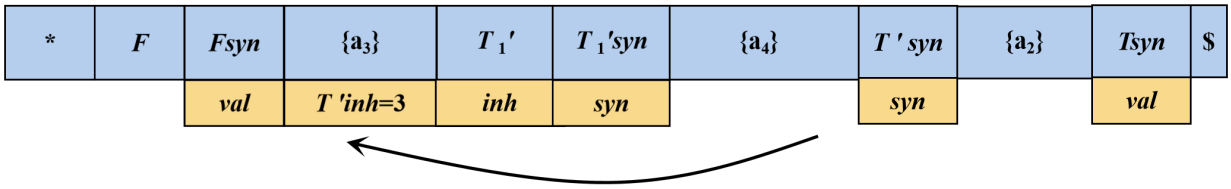


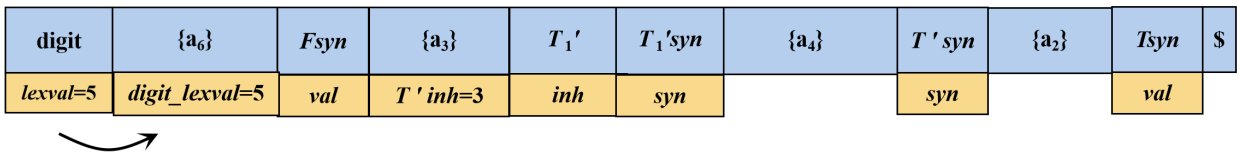
stack[top-1].val=stack[top].digit_lexval

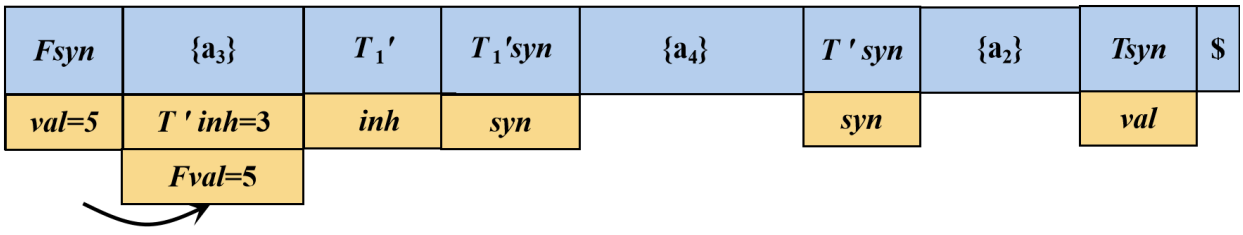


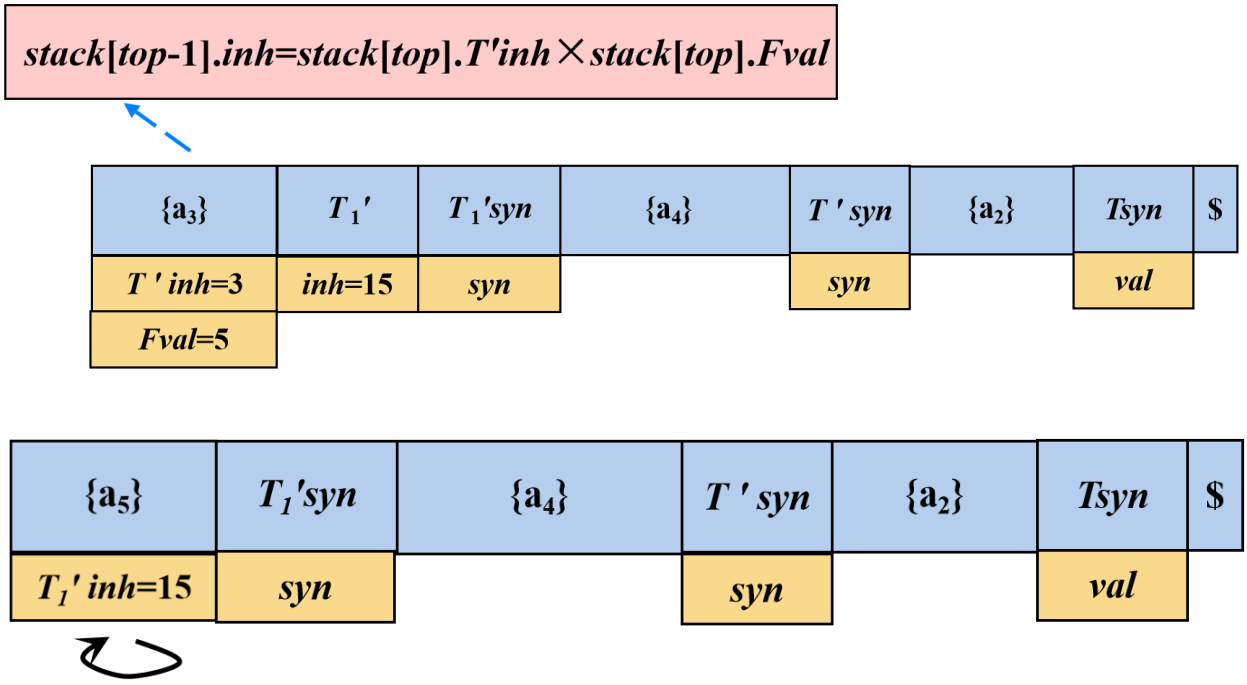

Fsyn
{a1}
T '
T ' syn
{a2}
Tsyn
$
val=3
Fval=3
inh
syn
val


stack[top-1].inh=stack[top].Fval
{a1}
T '
T ' syn
{a2}
Tsyn
$
Fval=3
inh=3
syn
val


*
F
Fsyn
{a3}
T 1'
T 1'syn
{a4}
T ' syn
{a2}
Tsyn
$
val
T 'inh=3
inh
syn
syn
val


digit
{a6}
Fsyn
{a3}
T 1'
T 1'syn
{a4}
T ' syn
{a2}
Tsyn
$
lexval=5
digit_lexval=5
val
T ' inh=3
inh
syn
syn
val


Fsyn
{a3}
T 1'
T 1'syn
{a4}
T ' syn
{a2}
Tsyn
$
val=5
T ' inh=3
Fval=5
inh
syn
syn
val



- T→M{a3}N{a4}
  - 在栈中先弹出T，再把**M，Msyn**,{a3}，**N，Nsyn**，{a4}加进去，之后可以根据栈中指针及相对位置来进行相关代码的书写和计算。
  - 如果栈顶（最左边）是**非终结符**，就按**展开式一直展开**直到匹配上终结符，都有**综合属性**，有**继承属性**的写在本身的下面。
  - 如果栈顶是**综合属性的属性值**Fsyn（综合记录出栈），也是要**弹出**，但要**看看后面是否会用到它**
  - 如果栈顶是**继承属性的属性值**F.inh（式子为F.inh = T.inh * F.inh），也是要弹出，会用到它的时候**把它放到对应的语义动作**下面
  - 如果栈顶是动作，就**执行**该动作

分析栈中的每一个记录都对应着一段执行代码

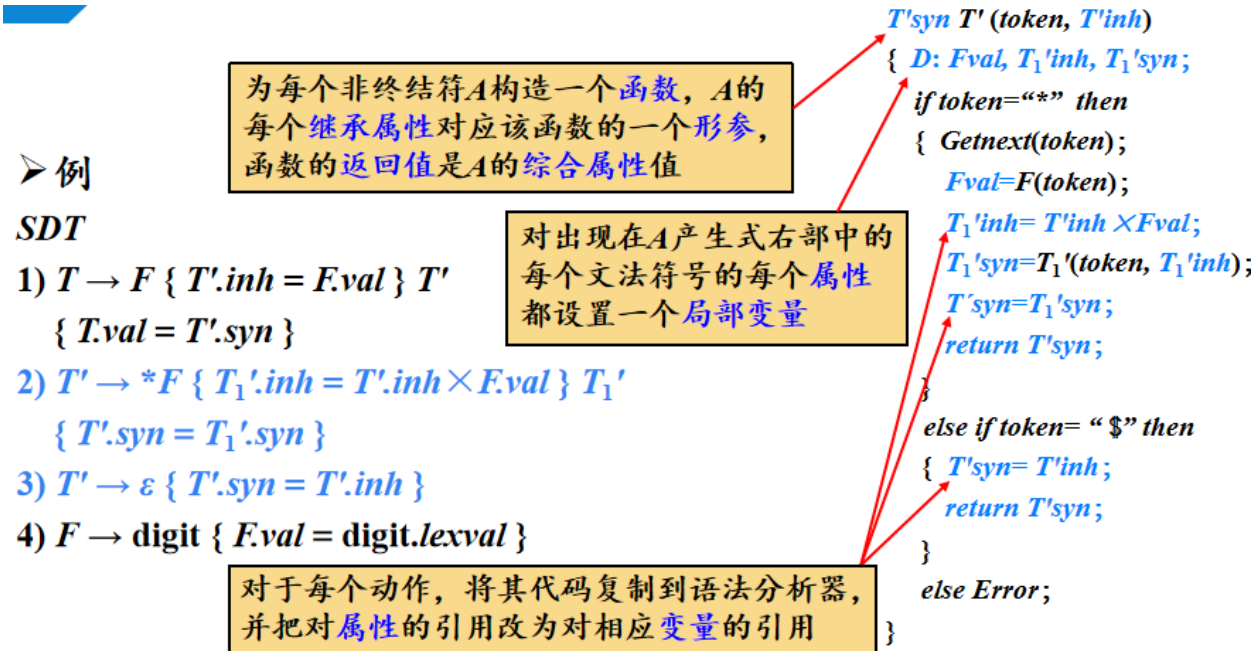
- 综合记录出栈时，要将综合属性值复制给后面特定的语义动作
- 变量展开时（即变量本身的记录出栈时），如果其含有继承属性，则要将继承属性值复制给后面特定的语义动作

1) $T \rightarrow F\ \{a_1\}\ T'\ \{a_2\}$	a_1: $T'.inh = F.val$
2) $T' \rightarrow *F\ \{a_3\}\ T_1'\ \{a_4\}$	a_2: $T.val = T'.syn$
3) $T' \rightarrow \varepsilon\ \{a_5\}$	a_3: $T_1'.inh = T'.inh \times F.val$
4) $F \rightarrow$ digit $\{a_6\}$	a_4: $T'.syn = T_1'.syn$
	a_5: $T'.syn = T'.inh$
	a_6: $F.val =$ digit.$lexval$

1) $T \rightarrow F\ \{a_1{:}T'.inh{=}F.val\}\ T'\ \{a_2{:}T.val{=}T'.syn\}$

符号	属性	执行代码
F		
$Fsyn$	val	$stack[top\text{-}1].Fval = stack[top].val$; $top=top\text{-}1$;
a_1	$Fval$	$stack[top\text{-}1].inh = stack[top].Fval$; $top=top\text{-}1$;
T'	inh	根据当前输入符号选择产生式进行推导 若选 2): $stack[top+3].T'inh = stack[top].inh$; $top=top+6$; 若选 3): $stack[top].T'inh = stack[top].inh$;
$T'syn$	syn	$stack[top\text{-}1].T'syn = stack[top].syn$; $top=top\text{-}1$;
a_2	$T'syn$	$stack[top\text{-}1].val = stack[top].T'syn$; $top=top\text{-}1$;

5.4.2 在递归的预测分析过程中进行翻译



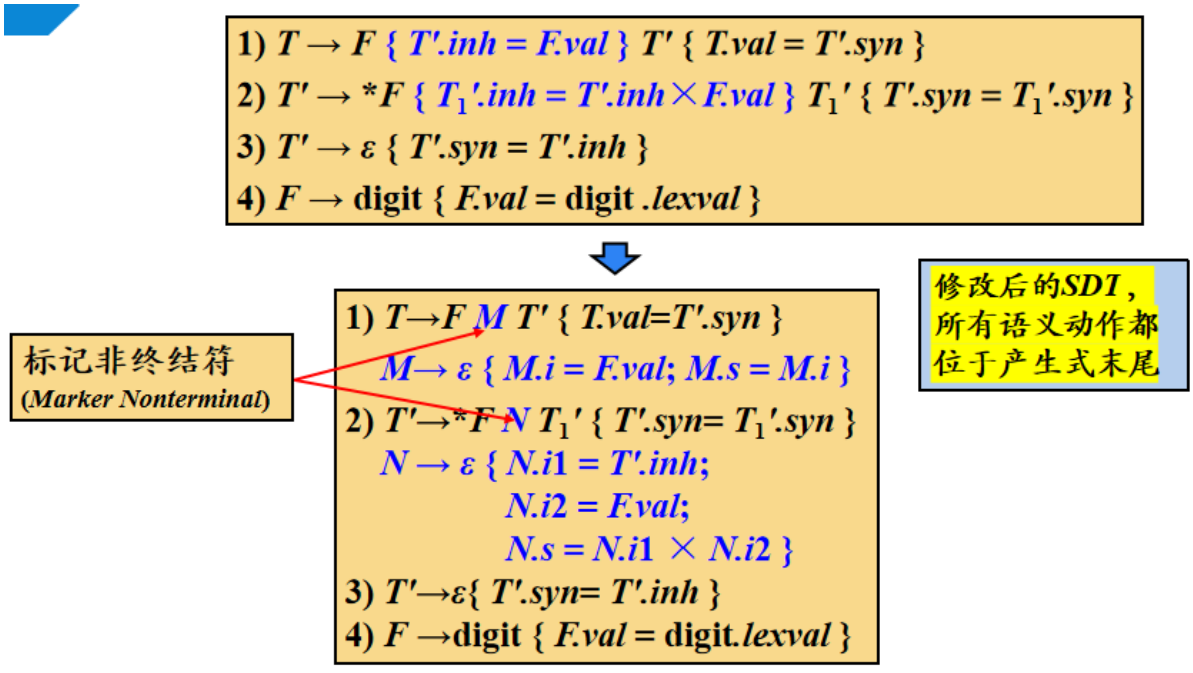
- **算法**
  - 为每个**非终结符**A构造一个**函数**
    - A的每个**继承属性**对应该函数的一个**形参**
    - 函数的**返回值**是A的**综合属性**值
    - 对出现在A产生式中的**每个文法符号的每个属性**都设置一个**局部变量**
  - **非终结符**A的代码根据**当前的输入**决定使用哪个产生式
  - 与每个产生式有关的代码执行如下动作：从左到右考虑产生式**右部**的**词法单元**、**非终结符**及**语义动作**
    - 对于**带有综合属性**x的**词法单元** X，把x的值保存在局部变量X.x中；然后产生一个匹配 X的调用，并继续输入
    - 对于**非终结符**B，产生一个右部**带有函数调用的赋值语句**c := B(b1 , b2 , ..., bk)，其中， b1 , b2 , ..., bk是代表B的**继承属性**的变量，c是代表B的**综合属性**的变量
    - 对于每个**语义动作**，将其代码复制到语法分析器，并把对属性的引用改为对相应变量的引用

5.5 L-属性定义的自底向上翻译

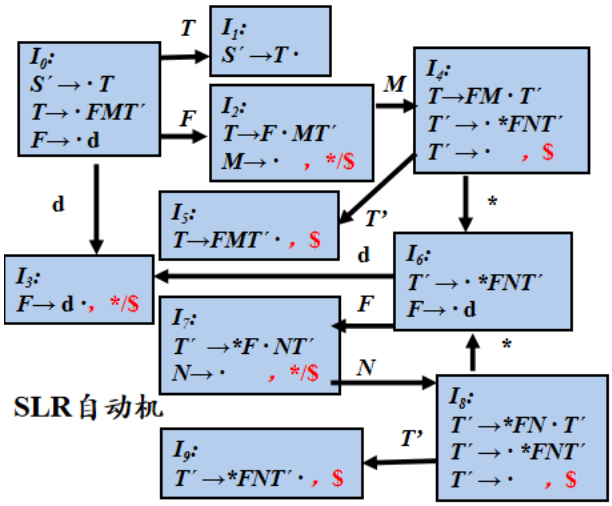
- 给定一个以**LL文法**为基础的L-SDD，可以**修改**这个**文法**，并在**LR语法分析**过程中计算这个新文法之上的SDD
  - 首先**构造SDT**，在各个**非终结符之前**放置语义动作来**计算它的继承属性**， 并在**产生式后端**放置语义动作计算**综合属性**
  - 对每个内嵌的**语义动作**，向文法中引入一个**标记非终结符**来替换它。每个这样的位置都有一个不同的标记，并且**对于任意一个标记M都有一个产生式M→ε**
  - 如果标记非终结符M在某个产生式`A→α{a}β`中替换了语义动作a，对a进行修改得到a' ，并且**将a'关联到M→ε 上**。动作a'

    (a) 将动作a需要的`A或α`中符号的任何属性作为**M的继承属性**进行复制

    (b) 按照a中的方法计算各个属性，但是将计算得到的这些属性作为M的**综合属性**



例



1) $T \rightarrow F\ M\ T'$ { $T.val = T'.syn$ }
   $M \rightarrow \varepsilon$ { $M.i = F.val$; $M.s = M.i$ }
2) $T' \rightarrow *F\ N\ T_1'$ { $T'.syn = T_1'.syn$ }
   $N \rightarrow \varepsilon$ { $N.i1 = T'.inh$;
   $N.i2 = F.val$;
   $N.s = N.i1 \times N.i2$ }
3) $T' \rightarrow \varepsilon$ { $T'.syn = T'.inh$ }
4) $F \rightarrow$ digit { $F.val$ = digit.$lexval$ }

输入：3 * 5
↑ ↑ ↑ ↑

0	2	4	6	7	8	9
$	F	M	*	F	N	T′
	3	T′inh=3		5	$T_1'inh$=15	syn=15

- **将语义动作改写为可执行的栈操作**

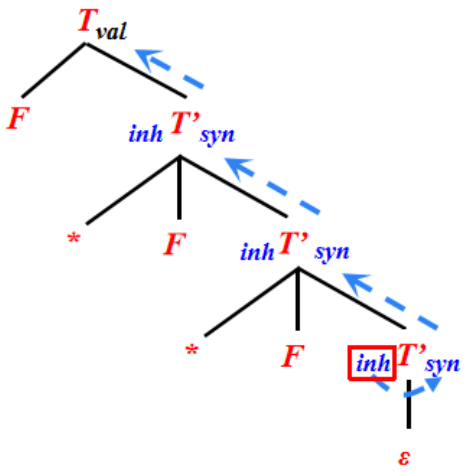
1) $T \rightarrow F\ M\ T'$ {$stack[top-2].val = stack[top].syn$; $top = top-2$;}
   $M \rightarrow \varepsilon$ {$stack[top+1].T'inh = stack[top].val$; $top = top+1$;}
2) $T' \rightarrow *F\ N\ T_1'$ {$stack[top-3].syn = stack[top].syn$; $top = top-3$;}
   $N \rightarrow \varepsilon$ {$stack[top+1].T'inh = stack[top-2].T'inh \times stack[top].val$; $top = top+1$;}
3) $T' \rightarrow \varepsilon$ {$stack[top+1].syn = stack[top].T'inh$; $top = top+1$;}
4) $F \rightarrow$ digit {$stack[top].val = stack[top].lexval$;}

- **L-SSD的设计思路分析**

例： *L-SDD*

	产生式	语义规则
(1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
(2)	$T' \rightarrow *\ F\ T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
(3)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
(4)	$F \rightarrow$ digit	$F.val =$ digit.*lexval*

语义翻译的主要任务：计算T的值



第六章 中间代码生成

6.1 声明语句的翻译

- 声明语句翻译的主要任务：收集**标识符的类型**等属性信息，并为每一个名字**分配一个相对地址**
  - 名字的**类型**和**相对地址信息**保存在相应的**符号表**记录中
- **类型表达式**
  - **基本类型**是类型表达式（eg.void）
  - 可以为类型表达式**命名**，**类型名**也是类型表达式
  - 将**类型构造符**作用于**类型表达式**可以构成新的类型表达式
    - 数组构造符array

      若T是类型表达式，则array (I, T)是类型表达式(I是一个整数)

类型	类型表达式
int [3]	*array* (3, *int*)
int [2][3]	*array* (2, *array*(3,*int*))

- 指针构造符pointer
  若T 是类型表达式，则 pointer (T) 是类型表达式，它表示一个指针类型
- 笛卡尔乘积构造符×
  若T1 和T2是类型表达式，则笛卡尔乘积T1 × T2 是类型表达式
- 函数构造符→
- 记录构造符record

➢ 设有C程序片段：

```
struct  stype
{  char[8] name;
   int  score;
};
stype[50] table;
stype* p;
```

➢ 和stype绑定的类型表达式
  ➢ record ((name×array(8, char)) × (score × integer))
➢ 和table绑定的类型表达式
  ➢ array (50, stype)
➢ 和p绑定的类型表达式
  ➢ pointer (stype)

- **局部变量的存储分配**
  - 从**类型表达式**可以知道该类型在**运行时刻**所需的**存储单元数量**称为**类型的宽度** (width)

- 在**编译时刻**，可以使用类型的宽度为每一个名字分配一个**相对地址**

变量声明语句的SDT

enter(name, type, offset):在符号表中为名字*name*创建记录,将*name*的类型设置为*type*,相对地址设置为*offset*

① $P \rightarrow \{ offset = 0 \}\ D$

② $D \rightarrow T$ id;$\{ enter(\text{id}.lexeme, T.type, offset);\ offset = offset + T.width; \}D$

③ $D \rightarrow \varepsilon$

④ $T \rightarrow B\ \{t = B.type; w = B.width;\}\ C\ \{ T.type = C.type; T.width = C.width; \}$

⑤ $T \rightarrow \uparrow T_1 \{ T.type = pointer(T_1.type); T.width = 4; \}$

⑥ $B \rightarrow$ int $\{ B.type = int; B.width = 4; \}$

⑦ $B \rightarrow$ real$\{ B.type = real; B.width = 8; \}$

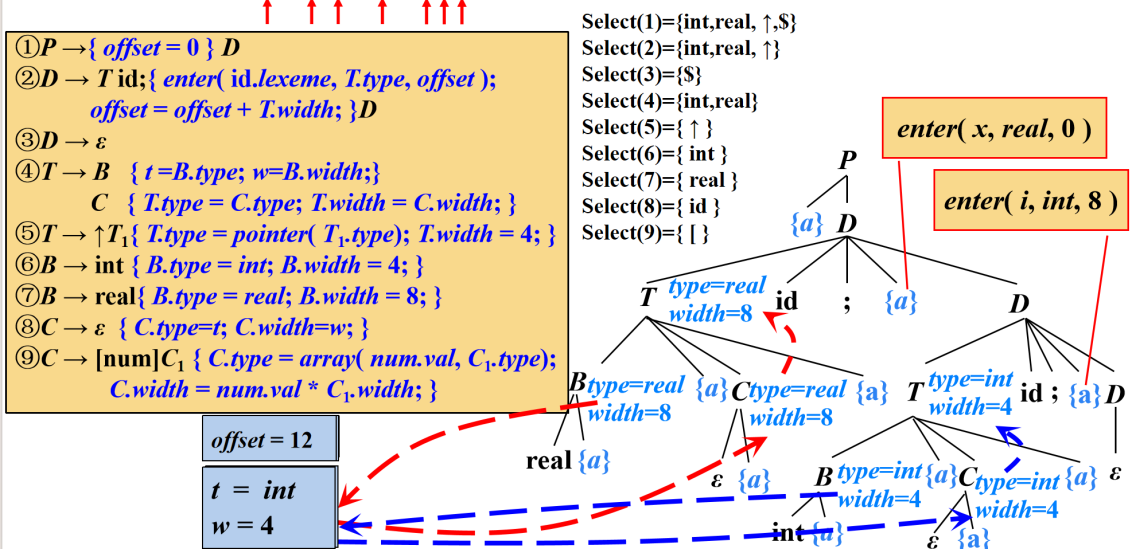
⑧ $C \rightarrow \varepsilon\ \{ C.type = t; C.width = w; \}$

⑨ $C \rightarrow [\text{num}]C_1 \{ C.type = array(num.val, C_1.type); C.width = num.val * C_1.width; \}$

符号	综合属性
B	*type, width*
C	*type, width*
T	*type, width*

变量	作用
offset	下一个可用的偏移地址
t, w	将类型和宽度信息从语法分析树中的*B*结点传递到对应于产生式$C \rightarrow \varepsilon$的结点

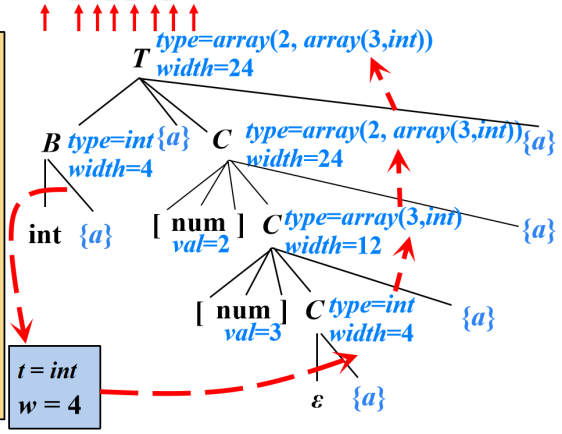
例: "*real x; int i;*"的语法制导翻译



PPT P13

例：数组类型表达式“*int*[2][3]”的语法制导翻译

①$P \rightarrow$ { *offset* = 0 } *D*
②$D \rightarrow T$ id;{ *enter*(id.*lexeme*, *T.type*, *offset*);
offset = *offset* + *T.width*; }*D*
③$D \rightarrow \varepsilon$
④$T \rightarrow B$ { *t* =*B.type*; *w*=*B.width*;}
C { *T.type* = *C.type*; *T.width* = *C.width*; }
⑤$T \rightarrow \uparrow T_1$ { *T.type* = *pointer*(T_1.*type*); *T.width* = 4; }
⑥$B \rightarrow$ int { *B.type* = *int*; *B.width* = 4; }
⑦$B \rightarrow$ real{ *B.type* = *real*; *B.width* = 8; }
⑧$C \rightarrow \varepsilon$ { *C.type*=*t*; *C.width*=*w*; }
⑨$C \rightarrow$ [num]C_1 { *C.type* = *array*(*num.val*, C_1.*type*);
C.width = *num.val* * C_1.*width*; }

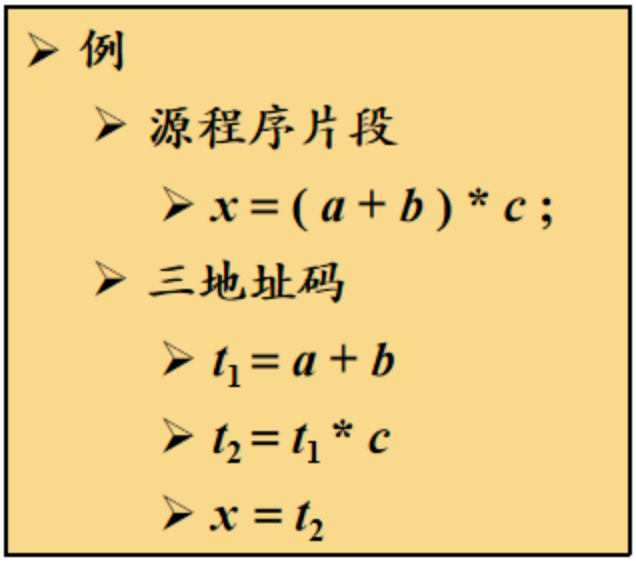


PPT P15

6.2 赋值语句的翻译

6.2.1 简单赋值语句的翻译

- 赋值语句翻译的主要任务
  - 生成对表达式求值的**三地址码**



赋值语句的*SDT*

符号	综合属性
S	*code*
E	*code* *addr*

$S \rightarrow$ id $= E$; { *p = lookup*(id.*lexeme*); *if p == nil then error* ;
S.code = E.code ||
gen(*p* '=' *E.addr*); }

$E \rightarrow E_1 + E_2$ { *E.addr = newtemp*( );
E.code = E_1*.code* || E_2*.code*||
gen(*E.addr* '=' E_1*.addr* '+' E_2*.addr*); }

newtemp( )：生成一个新的临时变量*t*，返回*t*的地址

$E \rightarrow E_1 * E_2$ { *E.addr = newtemp*( );
E.code = E_1*.code* || E_2*.code* ||
gen(*E.addr* '=' E_1*.addr* '*' E_2*.addr*); }

gen(*code*)：生成三地址指令*code*

$E \rightarrow -E_1$ { *E.addr = newtemp*( );
E.code = E_1*.code*||
gen(*E.addr* '=' '*uminus*' E_1*.addr*); }

$E \rightarrow (E_1)$ { *E.addr* = E_1*.addr*;
E.code= E_1*.code*; }

lookup(*name*)：查询符号表，返回*name* 对应的记录

$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*); *if E.addr == nil then error* ;
E.code = ''; }

增量翻译 (*Incremental Translation*)

$S \rightarrow$ id $= E$; { *p = lookup*(id.*lexeme*); *if p == nil then error* ;
gen(*p* '=' *E.addr*); }

在增量方法中，*gen*( )不仅要构造出一个新的三地址指令，还要将它添加到至今为止已生成的指令序列之后

$E \rightarrow E_1 + E_2$ { *E.addr = newtemp*( );
gen(*E.addr* '=' E_1*.addr* '+' E_2*.addr*); }

$E \rightarrow E_1 * E_2$ { *E.addr = newtemp*( );
gen(*E.addr* '=' E_1*.addr* '*' E_2*.addr*); }

$E \rightarrow -E_1$ { *E.addr = newtemp*( );
gen(*E.addr* '=' '*uminus*' E_1*.addr*); }

$E \rightarrow (E_1)$ { *E.addr* = E_1*.addr*;
}

$E \rightarrow$ id { *E.addr = lookup*(id.*lexeme*); *if E.addr == nil then error* ;
}

例

①$S \rightarrow id = E$; { *p = lookup(id.lexeme);*
if p==nil then error ;
gen(p '=' E.addr); }

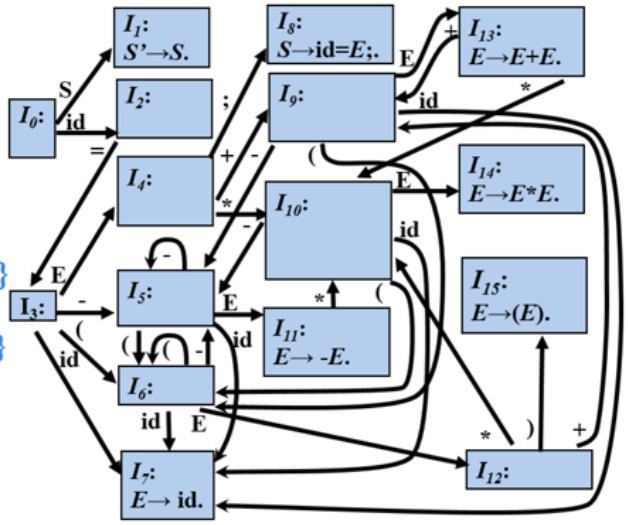
②$E \rightarrow E_1 + E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '+' E_2.addr); }

③$E \rightarrow E_1 * E_2$ { *E.addr = newtemp( );*
gen(E.addr '=' E_1.addr '' E_2.addr); }*

④$E \rightarrow -E_1$ { *E.addr = newtemp( );*
gen(E.addr '=' 'uminus' E_1.addr); }

⑤$E \rightarrow (E_1)$ { *E.addr = E_1.addr ; }*

⑥$E \rightarrow id$ { *E.addr = lookup(id.lexeme);*
if E.addr==nil then error ; }



例：$x = (a + b) * c$;

0	2	3	6	12	9	13
$	id	=	(	E	+	E
	x			a		b

$t_1 = a + b$

P23

6.2.2 数组引用的翻译

- 将数组引用翻译成三地址码时要解决的主要问题是**确定数组元素的存放地址**，也就是**数组元素的寻址**
- **数组元素寻址**

- **一维数组**
  - 假设每个数组元素的宽度是w，则数组元素$a[i]$的相对地址是：

    $base+i\times w$

    其中，$base$是数组的基地址，$i\times w$是偏移地址
- **二维数组**
  - 假设一行的宽度是w_1，同一行中每个数组元素的宽度是w_2，则数组元素$a[i_1]\ [i_2]$的相对地址是：

    $base+i_1\times w_1+i_2\times w_2$

    偏移地址
- **k维数组**
  - 数组元素$a[i_1]\ [i_2]\ \ldots[i_k]$的相对地址是：

    $base+i_1\times w_1+i_2\times w_2+\ldots+i_k\times w_k$

    偏移地址

$w_1 \rightarrow a[i_1]$ 的宽度
$w_2 \rightarrow a[i_1]\ [i_2]$ 的宽度
…
$w_k \rightarrow a[i_1]\ [i_2]\ \ldots[i_k]$的宽度

- 假设$type(a)= array(3, array(5, array(8, int)))$，一个整型变量占用4个字节，

$$addr(a[i_1][i_2][i_3]) = base + i_1 * w_1 + i_2*w_2 + i_3*w_3$$
$$= base + i_1*160 + i_2*32 + i_3*4$$

$a[i_1]$的宽度
$a[i_1][i_2]$的宽度
$a[i_1][i_2][i_3]$的宽度

- **带有数组引用的赋值语句的翻译**

➢例1

假设 $type(a)=array(n, int)$,

➢源程序片段

➢ $c = a[i]$;

➢三地址码

➢ $t_1 = i * 4$

➢ $t_2 = a[t_1]$

➢ $c = t_2$

$addr(a[i]) = base + i*4$

t_1

a表示数组的基地址

$S \rightarrow \text{id} = E;$ { $p = lookup(\text{id}.lexeme)$;
if p==nil then error ;
gen(p '=' E.addr); }

➢例2

假设 $type(a)= array(3, array(5, int))$,

➢源程序片段

➢ $c = a[i_1][i_2]$;

$addr(a[i_1][i_2])= base + i_1*20 + i_2*4$

t_1 t_2

➢三地址码

➢ $t_1 = i_1 * 20$

➢ $t_2 = i_2 * 4$

➢ $t_3 = t_1 + t_2$

➢ $t_4 = a[t_3]$

➢ $c = t_4$

- **数组元素寻址的SDT**

➢ 赋值语句的基本文法

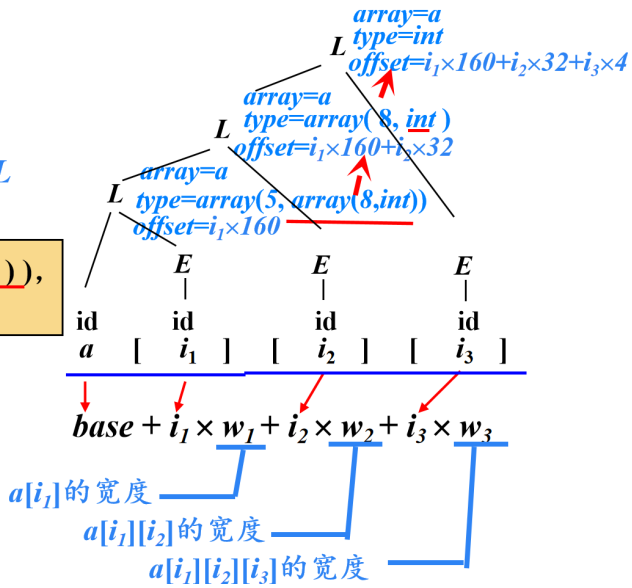
$S \rightarrow \mathbf{id} = E; \mid L = E;$

$E \rightarrow E_1 + E_2 \mid -E_1 \mid (E_1) \mid \mathbf{id} \mid L$

$L \rightarrow \mathbf{id}\ [E] \mid L_1\ [E]$

假设 *type*(*a*)= *array*(3, *array*(5, *array*(8, *int*))),
翻译语句片段 “*a*[i_1][i_2][i_3]”

- ➢ *L*的综合属性
  - ➢ *L.type*：*L*生成的数组元素的类型
  - ➢ *L.offset*：指示一个临时变量，该临时变量用于累加公式中的$i_j \times w_j$项，从而计算数组元素的偏移量
  - ➢ *L.array*：数组名在符号表的入口地址



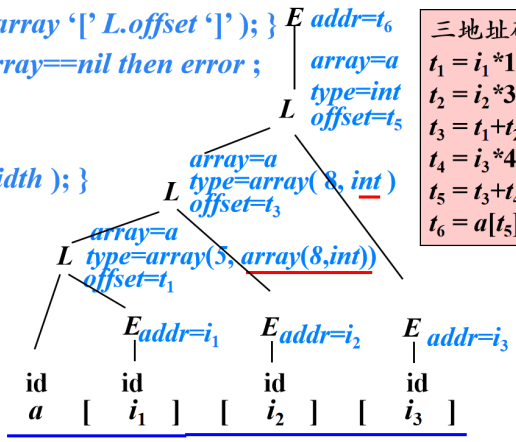
假设 *type*(*a*)= *array*(3, *array*(5, *array*(8, *int*))),
翻译语句片段 “*a*[i_1][i_2][i_3]”

$addr(a[i_1][i_2][i_3]) = base + i_1 \times w_1 + i_2 \times w_2 + i_3 \times w_3$
(t_1, t_2, t_4)

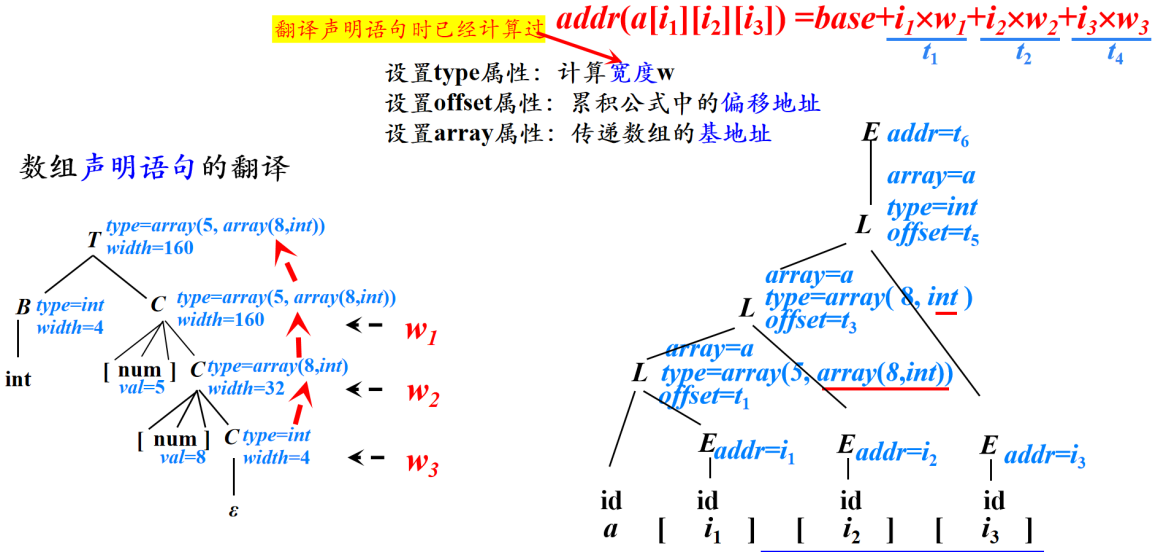
$S \rightarrow \mathbf{id} = E;$
  | *L* = *E*; { *gen*(*L.array* ‘[’ *L.offset* ‘]’ ‘=’ *E.addr*); }

$E \rightarrow E_1 + E_2 \mid -E_1 \mid (E_1) \mid \mathbf{id}$
  | *L* { *E.addr* = *newtemp*(); *gen*(*E.addr* ‘=’ *L. array* ‘[’ *L.offset* ‘]’); }

$L \rightarrow \mathbf{id}\ [E]$ { *L.array* = *lookup*(id.*lexeme*); *if L.array*==*nil then error* ;
  L.type = *L.array.type.elem* ;
  L.offset = *newtemp*();
  gen(*L.offset* ‘=’ *E.addr* ‘*’ *L.type.width*); }
  | L_1[*E*]{ *L.array* = L_1. *array*;
  L.type = L_1.*type.elem* ;
  t = *newtemp*();
  gen(*t* ‘=’ *E.addr* ‘*’ *L.type.width*);
  L.offset = *newtemp*();
  gen(*L.offset* ‘=’ L_1.*offset* ‘+’ *t*); }



三地址码
$t_1 = i_1 * 160$
$t_2 = i_2 * 32$
$t_3 = t_1 + t_2$
$t_4 = i_3 * 4$
$t_5 = t_3 + t_4$
$t_6 = a[t_5]$

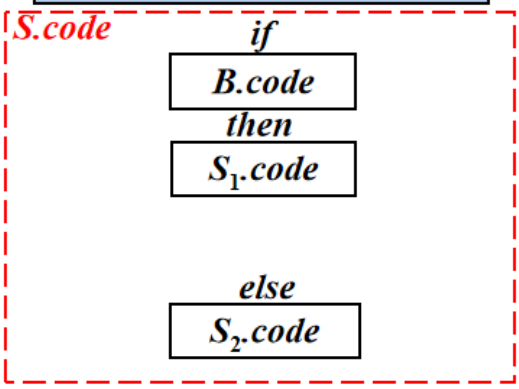


6.3 控制语句的翻译

- 控制流语句的基本文法
  - $P \rightarrow S$
  - $S \rightarrow S_1 S_2$
  - $S \rightarrow \text{id} = E ; \mid L = E ;$
  - $S \rightarrow \text{if } B \text{ then } S_1$
    $\mid \text{if } B \text{ then } S_1 \text{ else } S_2$
    $\mid \text{while } B \text{ do } S_1$

控制流语句的代码结构

➢ 例 $S \rightarrow if\ B\ then\ S_1\ else\ S_2$



例1：B="a<b"

三地址指令

if a<b goto S_1.first

goto S_2.first

标号(常量)

临时指令

if a<b goto B.true

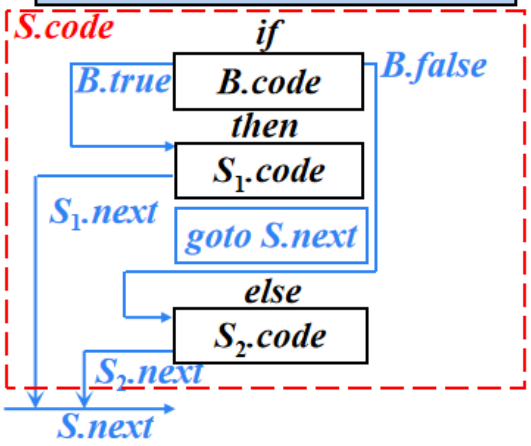
goto B.false

变量

布尔表达式B被翻译成由跳转指令构成的跳转代码

用指令的标号标识一条三地址指令

➢ 例 $S \rightarrow if\ B\ then\ S_1\ else\ S_2$



布尔表达式B被翻译成由跳转指令构成的跳转代码

➢ 继承属性

- ***B.true***：是一个地址，该地址用来存放当*B*为真时控制流转向的指令的标号
- ***B.false***：是一个地址，该地址用来存放当*B*为假时控制流转向的指令的标号
- ***S.next***：是一个地址，该地址用来存放紧跟在*S*代码之后执行的指令(*S*的后继指令)的标号

用指令的标号标识一条三地址指令

控制流语句的SDT

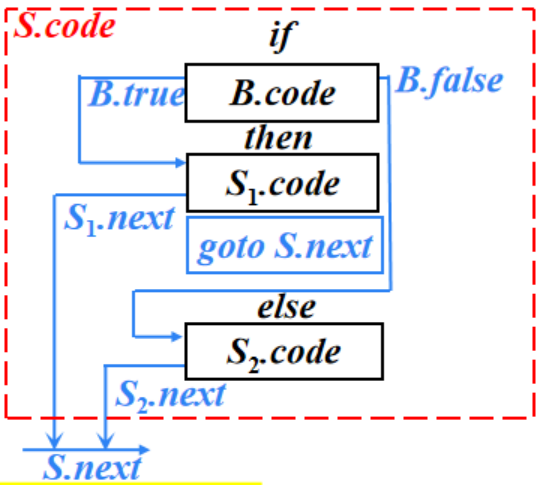
newlabel( ): 生成一个用于存放标号的新的临时变量L，返回变量地址

- $P \rightarrow \{ S.next = newlabel(); \} S \{ label(S.next); \}$
- $S \rightarrow \{ S_1.next = newlabel(); \} S_1$
  $\{ label(S_1.next); S_2.next = S.next ; \} S_2$

label(L): 将下一条三地址指令的标号存放到地址L中

if-then-else语句的SDT

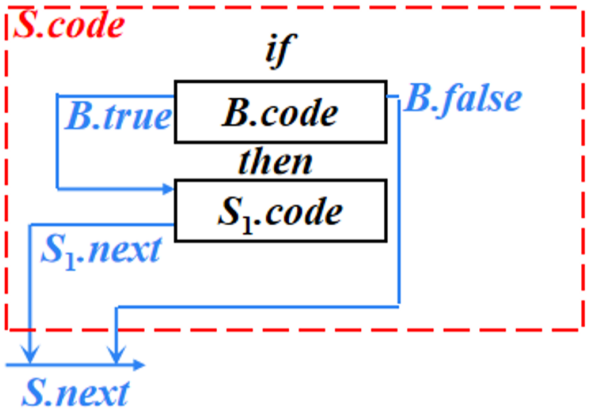
$S \rightarrow if\ B\ then\ S_1\ else\ S_2$



$S \rightarrow if \{ B.true = newlabel(); \ B.false = newlabel(); \} B$
$then \{ S_1.next = S.next; \ label(B.true); \} S_1 \ \{ gen(\text{'goto'}\ S.next) \}$
$else \ \{ S_2.next = S.next; \ label(B.false); \} S_2$

if-then语句的SDT

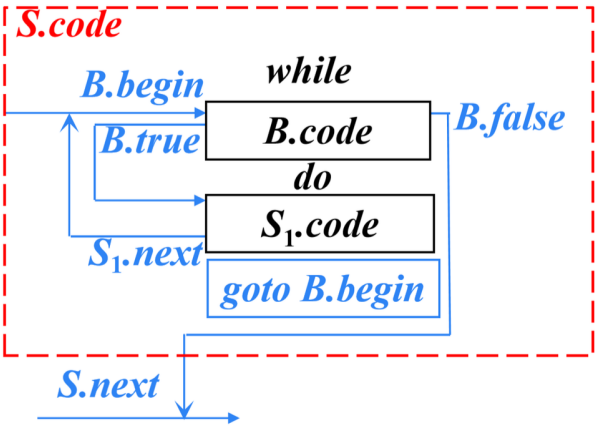
$S \rightarrow if\ B\ then\ S_1$



$S \rightarrow if\ \{B.true = newlabel();\ \ B.false = S.next;\ \}\ B$
$then\ \{S_1.next = S.next;\ \ label(B.true);\ \ \}\ S_1$

while-do语句的SDT

$S \rightarrow while\ B\ do\ S_1$



$S \rightarrow while\ \{\ B.true = newlabel();$
$B.false = S.next;$
$B.begin = newlabel();$
$label(B.begin);\ \ \ \ \}\ B$
$do\ \{\ S_1.next = B.begin;\ \ label(B.true);\ \}\ S_1$
$\{\ gen(\text{‘goto’}\ B.begin);\ \}$

控制流语句SDT编写要点

- 分析每一个非终结符之前
  - 先计算继承属性
  - 再观察代码结构图中该非终结符对应的方框顶部是否有导入箭头。如果有，调用label( )函数
- 上一个代码框执行完不顺序执行下一个代码框时，生成一条显式跳转指令
- 有自下而上的箭头时，设置begin属性。且定义后直接调用label( )函数绑定地址

布尔表达式的翻译

- 布尔表达式的基本文法

$B \rightarrow B$ or B
| B and B
| not B
| (B)
| E relop E
| true
| false

逻辑运算符优先级：not > and > or

关系表达式

relop（关系运算符）：
<, <=, >, >=, ==, !=

- 在跳转代码中，逻辑运算符&&、|| 和！被翻译成跳转指令。运算符本身不出现在代码中，布尔表达式的值是通过代码序列中的位置来表示的
- 例
  - 语句

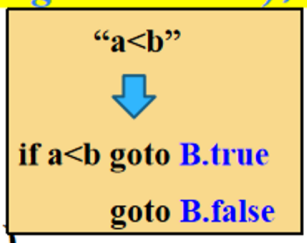
```
if ( x<100 || x>200 && x!=y )
        x=0;
```

  - 三地址代码

```
        if x<100 goto L2
        goto L3
L3 :    if x>200 goto L4
        goto L1
L4 :    if x!=y goto L2
        goto L1
L2 :    x=0
L1 :
```

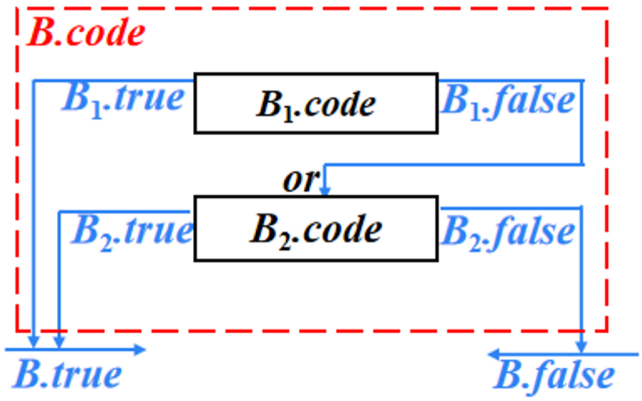
布尔表达式的SDT

- $B \rightarrow E_1$ relop E_2 { *gen('if' E_1.addr relop E_2.addr 'goto' B.true);*
  gen('goto' B.false); }
- $B \rightarrow$ true { *gen('goto' B.true);* }
- $B \rightarrow$ false { *gen('goto' B.false);* }
- $B \rightarrow$ ({ B_1.true = B. true; B_1.false = B.false; } B_1)
- $B \rightarrow$ not { B_1.true = B.false; B_1.false = B.true; } B_1



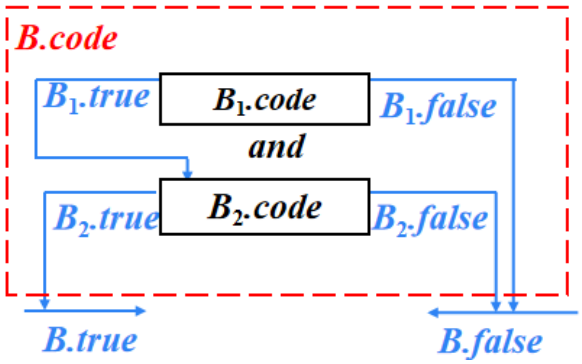
B→B1 or B2的SDT

- $B \rightarrow B_1 \ or \ B_2$
  - $B \rightarrow \{ B_1.true = B.true;\ B_1.false = newlabel();\ \} B_1$
    or $\{B_2.true = B.true;\ B_2.false = B.false;\ label(B_1.false);\ \} B_2$



B → B1 and B2 的SDT

- $B \rightarrow B_1 \ and \ B_2$
  - $B \rightarrow \{ B_1.true = newlabel();\ B_1.false = B.false;\ \} B_1$
    and $\{B_2.true = B.true;\ B_2.false = B.false;\ label(B_1.true);\ \} B_2$



控制流语句的*SDT*

SDD：L-SDD
- 赋值语句：只定义了综合属性
- 分支、循环语句：只定义了继承属性，且不依赖右兄弟节点属性值

基础文法：可以使用LR分析技术

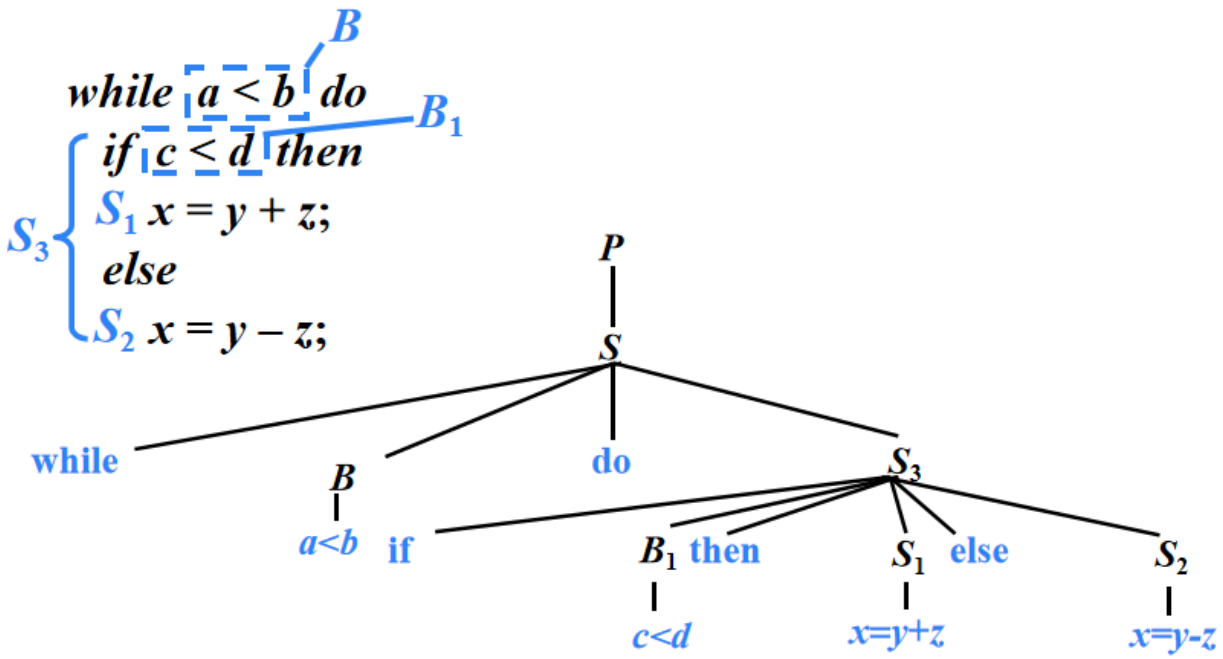
- $P \rightarrow \{a\}S\{a\}$
- $S \rightarrow \{a\}S_1\{a\}S_2$
- $S \rightarrow \text{id}=E;\{a\} \mid L=E;\{a\}$
- $E \rightarrow E_1+E_2\{a\} \mid -E_1\{a\} \mid (E_1)\{a\} \mid \text{id}\{a\} \mid L\{a\}$
- $L \rightarrow \text{id}[E]\{a\} \mid L_1[E]\{a\}$
- $S \rightarrow$ if $\{a\}B$ then $\{a\}S_1$
  | if $\{a\}B$ then $\{a\}S_1$ else $\{a\}S_2$
  | while $\{a\}B$ do $\{a\}S_1\{a\}$
- $B \rightarrow \{a\}B$ or $\{a\}B \mid \{a\}B$ and $\{a\}B \mid$ not $\{a\}B \mid (\{a\}B)$
  | E relop $E\{a\} \mid$ true$\{a\} \mid$ false$\{a\}$

例：

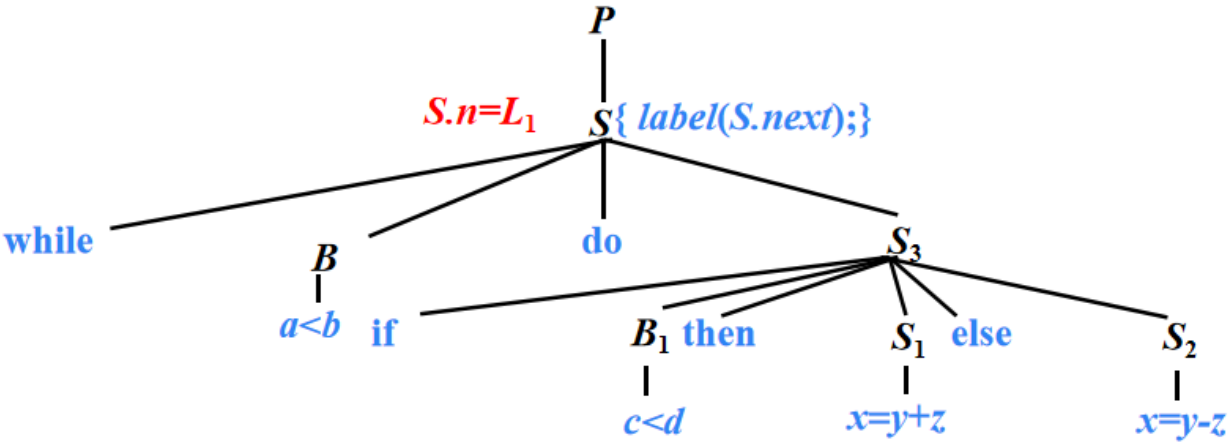
```
while a < b do
  if c < d then
    x = y + z;
  else
    x = y – z;
```

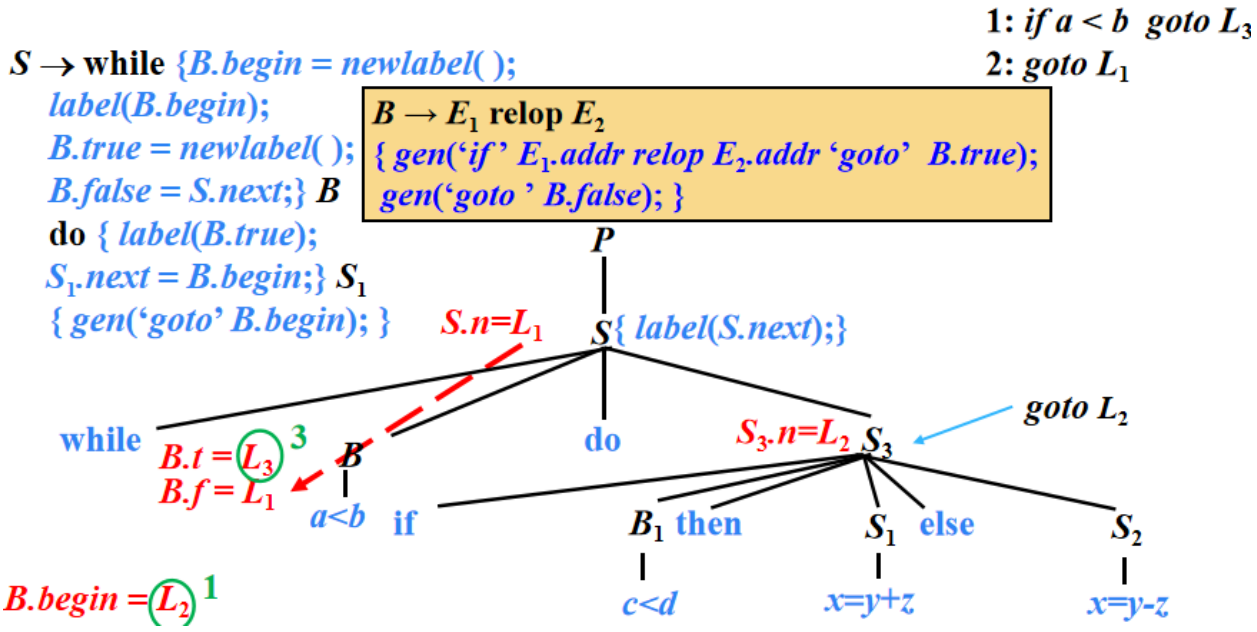
*SDT*的通用实现方法

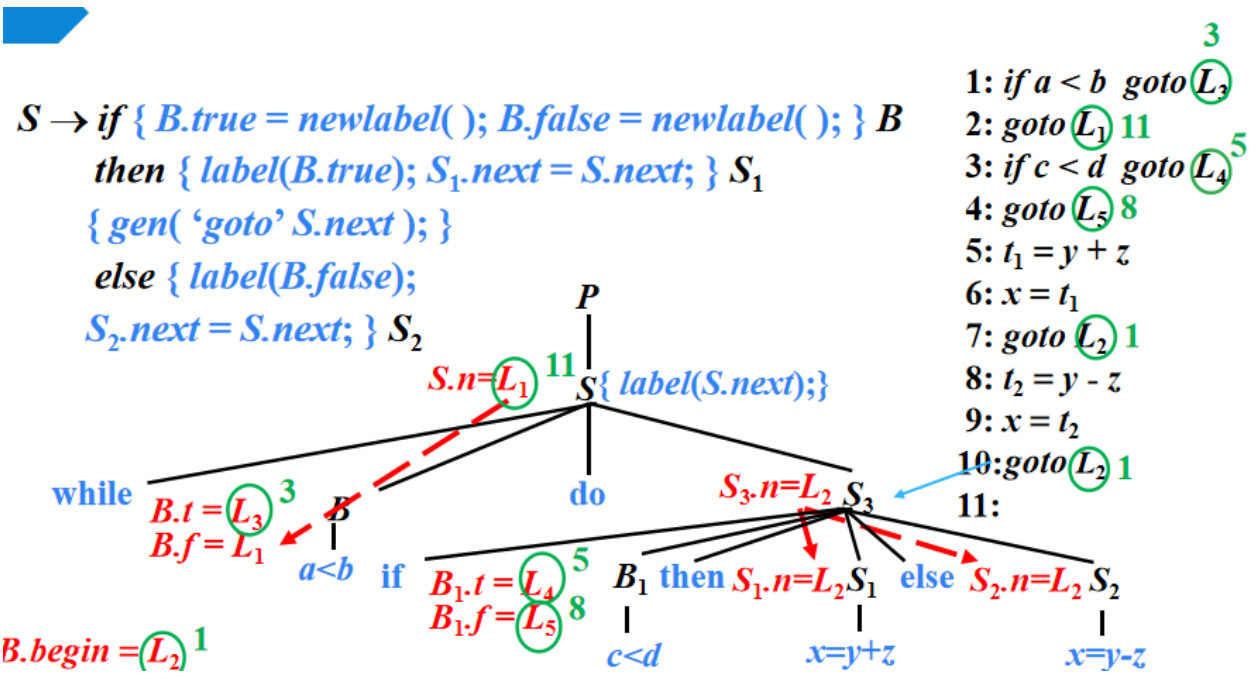
- 任何*SDT*都可以通过下面的方法实现
  - 首先建立一棵语法分析树
  - 然后按照从左到右的深度优先顺序来执行这些动作



$$P \rightarrow \{\ S.next = newlabel(\ );\ \}S\{\ label(S.next);\}$$




S → while {B.begin = newlabel( );
label(B.begin);
B.true = newlabel( );
B.false = S.next;} B
do { label(B.true);
S1.next = B.begin;} S1
{ gen('goto' B.begin); }
B → E1 relop E2
{ gen('if' E1.addr relop E2.addr 'goto' B.true);
gen('goto ' B.false); }
1: if a < b goto L3
2: goto L1
P
S.n=L1
S{ label(S.next);}
while
B.t = L3
B.f = L1
3
B
a<b
if
do
S3.n=L2 S3
goto L2
B1 then
S1
else
S2
c<d
x=y+z
x=y-z
B.begin = L2
1


S → if { B.true = newlabel( ); B.false = newlabel( ); } B
then { label(B.true); S1.next = S.next; } S1
{ gen('goto' S.next); }
else { label(B.false);
S2.next = S.next; } S2
1: if a < b goto L3
2: goto L1
3: if c < d goto L4
4: goto L5
5: t1 = y + z
6: x = t1
7: goto L2
8: t2 = y - z
9: x = t2
10:goto L2
11:
3
11
5
8
1
P
S.n=L1
S{ label(S.next);}
while
B.t = L3
B.f = L1
3
B
a<b
if
do
S3.n=L2 S3
B1.t = L4
B1.f = L5
5
8
B1 then
S1.n=L2 S1
else
S2.n=L2 S2
c<d
x=y+z
x=y-z
B.begin = L2
1

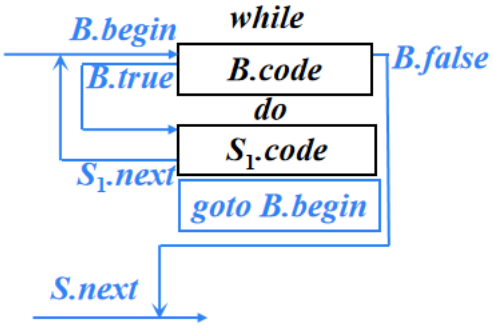
语句 “*while a<b do if c<d then x=y+z else x=y-z*” 的三地址代码

1: *if a < b goto 3*	1: (*j<*, *a*, *b* , 3)
2: *goto* 11	2: (*j* , - , - , 11)
3: *if c < d goto* 5	3: (*j<*, *c*, *d* , 5)
4: *goto* 8	4: (*j* , - , - , 8)
5: $t_1 = y + z$	5: (+ , *y* , *z* , t_1)
6: $x = t_1$	6: (= , t_1, - , *x*)
7: *goto* 1	7: (*j* , - , - , 1)
8: $t_2 = y - z$	8: (- , *y*, *z* , t_2)
9: $x = t_2$	9: (= , t_2, - , *x*)
10: *goto* 1	10: (*j* , - , - , 1)
11:	11:

- 优化while

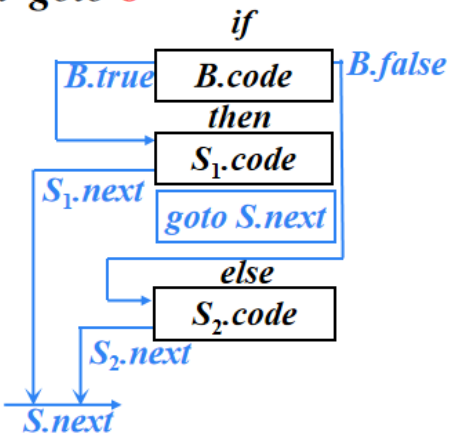
B.first →	1: *if a < b goto* 3	1: *ifFalse a < b goto* 11
	2: *goto* 11	2:
S_1*.first* →	3: *if c < d goto* 5	3: *if c < d goto* 5
	4: *goto* 8	4: *goto* 8
	5: $t_1 = y + z$	5: $t_1 = y + z$
	6: $x = t_1$	6: $x = t_1$
	7: *goto* 1	7: *goto* 1
	8: $t_2 = y - z$	8: $t_2 = y - z$
	9: $x = t_2$	9: $x = t_2$
	10: *goto* 1	10: *goto* 1
	11:	11:

避免生成冗余的goto指令

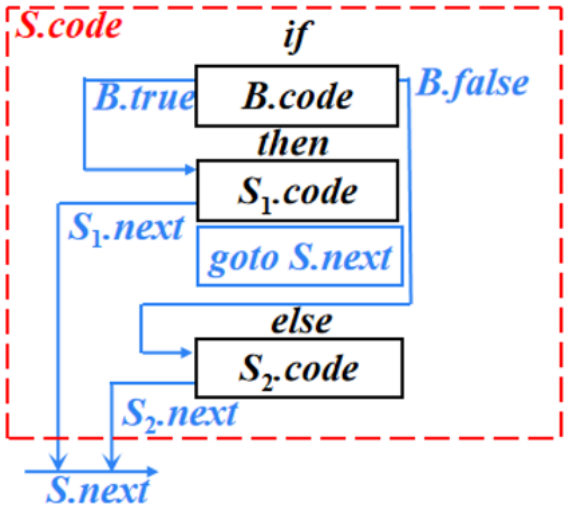


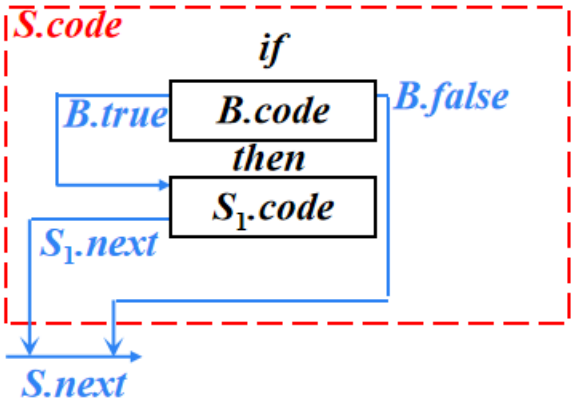
- 优化if-else

	1: *if* $a < b$ *goto* 3	1: *ifFalse* $a < b$ *goto* 11
	2: *goto* 11	2:
B.first →	3: *if* $c < d$ *goto* 5	3: *ifFalse* $c < d$ *goto* 8
	4: *goto* 8	4:
S_1*.first* →	5: $t_1 = y + z$	5: $t_1 = y + z$
	6: $x = t_1$	6: $x = t_1$
	7: *goto* 1	7: *goto* 1
S_2*.first* →	8: $t_2 = y - z$	8: $t_2 = y - z$
	9: $x = t_2$	9: $x = t_2$
	10: *goto* 1	10: *goto* 1
	11:	11:



避免生成冗余的goto语句





$$S \rightarrow if \{ B.true = newlabel(); B.false = S.next; \} B \; then \{ S_1.next = S.next; label(B.true); \} S_1$$

省略。不生成任何跳转指令

$$S \rightarrow if \{ B.true = fall; B.false = S.next; \} B \; then \{ S_1.next = S.next; \} S_1$$

修改B → E1 relop E2的SDT

$$B \rightarrow E_1 \text{ relop } E_2 \{ gen(\text{'if'} \; E_1.addr \; relop \; E_2.addr \; \text{'goto'} \; B.true); \; gen(\text{'goto'} \; B.false); \}$$

$$B \rightarrow E_1 \text{ relop } E_2$$

{ if $B.true \neq fall$ and $B.false \neq fall$ then
$gen(\text{'if'} \; E_1.addr \; relop \; E_2.addr \; \text{'goto'} \; B.true);$ $gen(\text{'goto'} \; B.false);$ }
else if $B.true \neq fall$ then $gen(\text{'if'} \; E_1.addr \; relop \; E_2.addr \; \text{'goto'} \; B.true);$
else if $B.false \neq fall$ then $gen(\text{'ifFalse'} \; E_1.addr \; relop \; E_2.addr \; \text{'goto'} \; B.false);$
else ' ' }

1. **条件判断**：
   - **B.true ≠ fall and B.false ≠ fall**：如果`B.true`和`B.false`都不为`fall`，表示它们都有对应的跳转位置。这时，生成原始规则中的两条指令。

     ```
     gen('if' E1.addr relop E2.addr 'goto' B.true);
     gen('goto' B.false);
     ```

2. **仅B.true不为fall**：
   - **B.true ≠ fall**：如果只有`B.true`有跳转位置，而`B.false`没有，则只需生成一条条件跳转指令，指向`B.true`。如果条件为假，自然会执行后续代码。

```
gen('if' E1.addr relop E2.addr 'goto' B.true);
```

3. **仅B.false不为fall**：
   - **B.false ≠ fall**：如果只有`B.false`有跳转位置，而`B.true`没有，则需要生成一条反向条件跳转指令`ifFalse`，指向`B.false`。

     ```
     gen('ifFalse' E1.addr relop E2.addr 'goto' B.false);
     ```

4. **都为fall**：
   - 如果`B.true`和`B.false`都为`fall`，表示没有具体的跳转位置，这种情况下不需要生成任何指令。

修改B → B1 or B2 的SDT

$$B \rightarrow \{ B_1.true = B.true;\ B_1.false = newlabel(); \}B_1 \text{ or } \{ B_2.true = B.true;\ B_2.false = B.false; label(B_1.false); \}B_2$$

$$B \rightarrow \{ B_1.true = B.true == fall\ ?\ newlabel() : B.true; B_1.false = fall; \}B_1 \text{ or } \{B_2.true = B.true; B_2.false = B.false; \}B_2\ \{\text{if } B.true == fall \text{ then } label(B_1.true);\}$$

B1 部分的修改

- `B1.true = B.true == fall ? newlabel() : B.true`：
  - 如果`B.true`为`fall`（即没有具体的跳转位置），则为`B1.true`创建一个新的标签。
  - 否则，将`B1.true`设置为`B.true`的跳转位置。
- `B1.false = fall`：将`B1.false`设置为`fall`，表示没有具体的跳转位置。

B2 部分的修改

- `B2.true = B.true`：设置`B2`的`true`跳转位置为`B`的`true`跳转位置。
- `B2.false = B.false`：设置`B2`的`false`跳转位置为`B`的`false`跳转位置。

最后一个部分

- `if B.true == fall then label(B1.true)`：
  - 如果 `B.true` 为 `fall`，则在 `B1.true` 的标签位置插入代码。

修改B → B1 and B2 的SDT

$$B \rightarrow \{ B_1.true = newlabel(); B_1.false = B.false; \} B_1 \text{ and } \{B_2.true = B.true; B_2.false = B.false; label(B_1.true); \} B_2$$

$$B \rightarrow \{ B_1.true = fall; B_1.false = B.false == fall\ ?\ newlabel() : B.false; \} B_1 \text{ and } \{ B_2.true = B.true; B_2.false = B.false; \} B_2 \{\text{if } B.false == fall \text{ then } label(B_1.false); \}$$

例

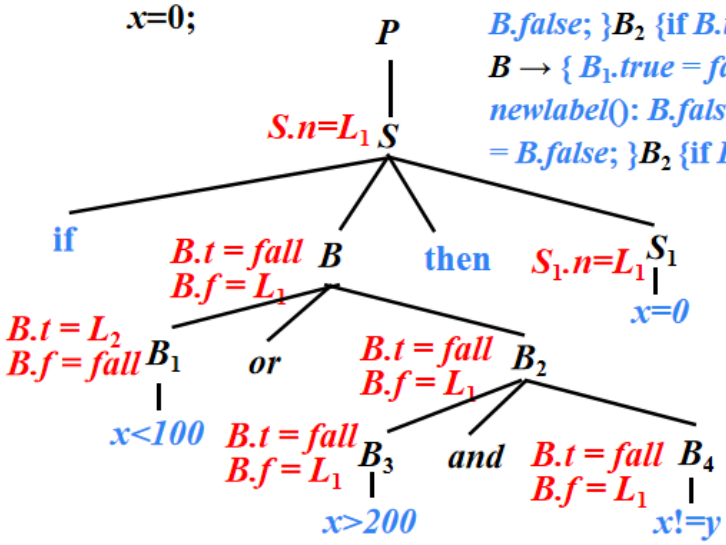
$S \rightarrow$ *if* $\{ B.true = fall; B.false = S.next; \} B$ *then* $\{ S_1.next = S.next; \} S_1$

$B \rightarrow \{ B_1.true = B.true == fall\ ?\ newlabel() : B.true; B_1.false = fall; \} B_1$ or $\{B_2.true = B.true;\ B_2.false = B.false; \} B_2 \{\text{if } B.true == fall \text{ then } label(B_1.true); \}$

$B \rightarrow \{ B_1.true = fall; B_1.false = B.false == fall\ ?\ newlabel() : B.false; \} B_1$ and $\{B_2.true = B.true; B_2.false = B.false; \} B_2 \{\text{if } B.false == fall \text{ then } label(B_1.false); \}$

if ($x<100 \ || \ x>200 \ \&\& \ x!=y$)

$x=0$;



if $x<100$ *goto* L_2

ifFalse $x>200$ *goto* L_1

ifFalse $x!=y$ *goto* L_1

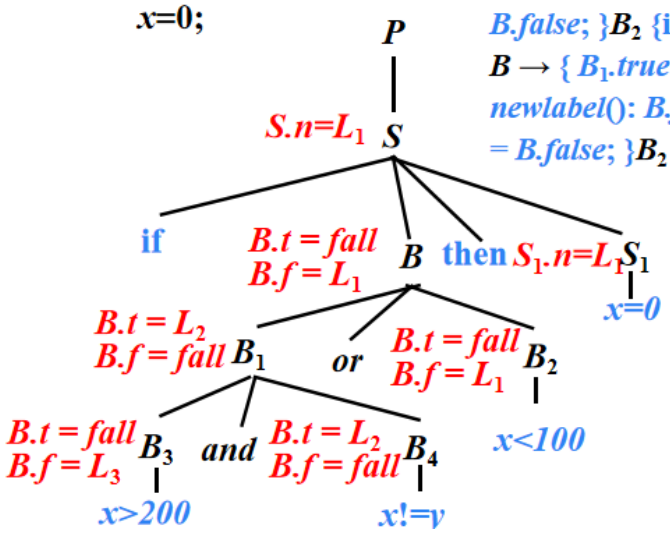
L_2 : $x=0$

L_1 :

例

if (*x*>200 && *x*!=*y* || *x*<100)

x=0;



$S \rightarrow$ *if* { *B.true* = *fall*; *B.false* = *S.next*; } *B*
then { S_1.*next* = *S.next*; } S_1

$B \rightarrow$ { B_1.*true* = *B.true*==*fall*? *newlabel*():*B.true*; B_1.*false* = *fall*; }B_1 or {B_2.*true* = *B.true*; B_2.*false* = *B.false*; }B_2 {if *B.true* ==*fall* then *label*(B_1.*true*);}

$B \rightarrow$ { B_1.*true* = *fall*; B_1.*false* = *B.false*==*fall* ? *newlabel*(): *B.false*;} B_1 and {B_2.*true* = *B.true*; B_2.*false* = *B.false*; }B_2 {if *B. false* ==*fall* then *label*(B_1.*false*);}

ifFalse x>200 *goto* L_3
if x!=*y goto* L_2
L_3 : *ifFalse x*<100 *goto* L_1
L_2 : *x*=0
L_1 :

P73

6.4 回填

- 基本思想
  - 生成一个跳转指令时，暂时不指定该跳转指令的目标标号。这样的指令都被放入由跳转指令组成的列表中。同一个列表中的所有跳转指令具有相同的目标标号。等到能够确定正确的目标标号时，才去填充这些指令的目标标号

非终结符B的综合属性

- $B.truelist$：指向一个包含跳转指令的列表，这些指令最终获得的目标标号就是当B为真时控制流应该转向的指令的标号
- $B.falselist$：指向一个包含跳转指令的列表，这些指令最终获得的目标标号就是当B为假时控制流应该转向的指令的标号

函数

- $makelist(i)$
  - 创建一个只包含i的列表，i是跳转指令的标号，函数返回指向新创建的列表的指针
- $merge(p_1, p_2)$
  - 将p_1和p_2指向的列表进行合并，返回指向合并后的列表的指针
- $backpatch(p, i)$ 回填
  - 将i作为目标标号插入到p所指列表中的各指令中

布尔表达式的回填

➢ $B \rightarrow E_1$ relop E_2

{

B.truelist = makelist(nextquad);

B.falselist = makelist(nextquad+1);

gen('if' E_1.addr relop E_2.addr 'goto _');

gen('goto _');

}

nextquad：即将生成的下一条指令的标号

➢ $B \rightarrow$ true

{

B.truelist = makelist(nextquad);

gen('goto _');

}

➢ $B \rightarrow$ false

{

B.falselist = makelist(nextquad);

gen('goto _');

}

➢ $B \rightarrow (B_1)$

{

$B.truelist = B_1.truelist$;

$B.falselist = B_1.falselist$;

}

➢ $B \rightarrow \text{not } B_1$

{

$B.truelist = B_1.falselist$;

$B.falselist = B_1.truelist$;

}

B → B1 or B2

$B \rightarrow B_1 \text{ or } M\ B_2$

M：$B_1.falselist$的跳转目标是B2，想要加入的动作

{

$B.truelist = merge(B_1.truelist, B_2.truelist)$;

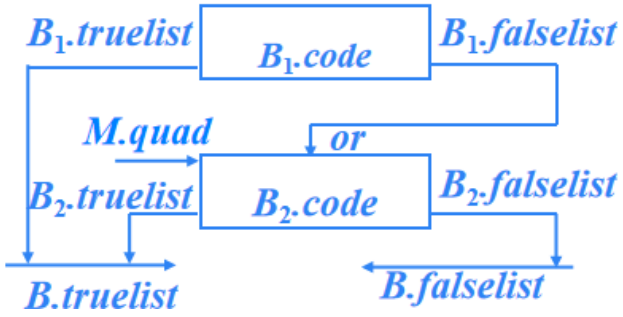
$B.falselist = B_2.falselist$;

$backpatch(B_1.falselist, M.quad)$;

}

$M \rightarrow \varepsilon$

{ $M.quad = nextquad$; }



B → B1 and B2

$B \rightarrow B_1$ and M B_2

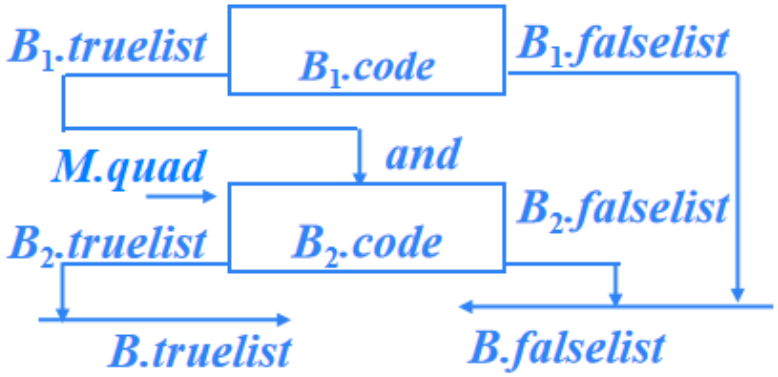
{

$B.truelist = B_2.truelist;$

$B.falselist = merge(B_1.falselist, B_2.falselist);$

$backpatch(B_1.truelist, M.quad);$

}



例

$B \rightarrow B_1$ or M B_2

{

backpatch(B_1.falselist, M.quad);

B.truelist = merge(B_1.truelist, B_2.truelist);

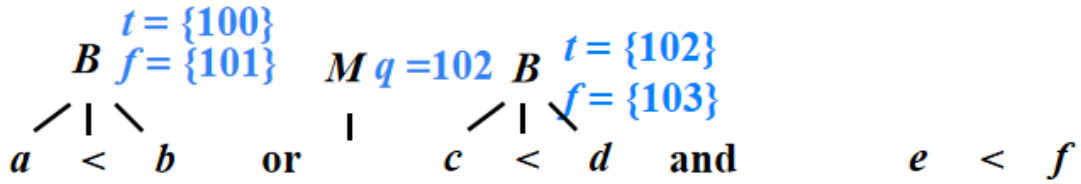
B.falselist = B_2.falselist ;

}

$M \rightarrow \varepsilon$

{ *M.quad = nextquad ;* }

100: *if a<b goto _*
101: *goto _*
102: *if c<d goto _*
103: *goto _*



P86

$B \rightarrow B_1$ and M B_2

{

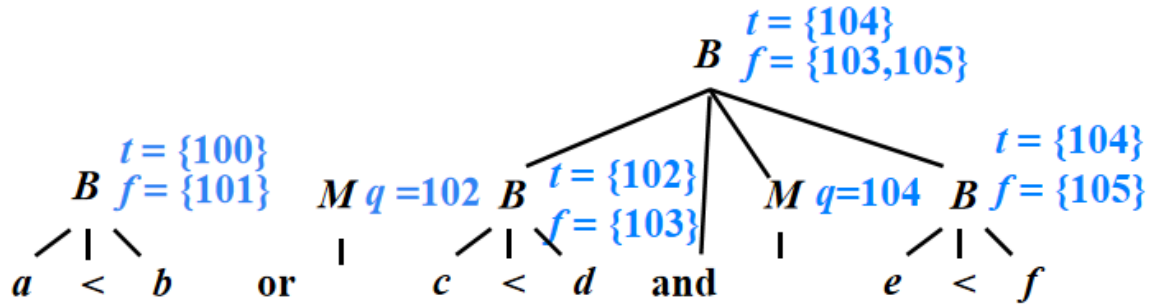
B.truelist = B_2.truelist;

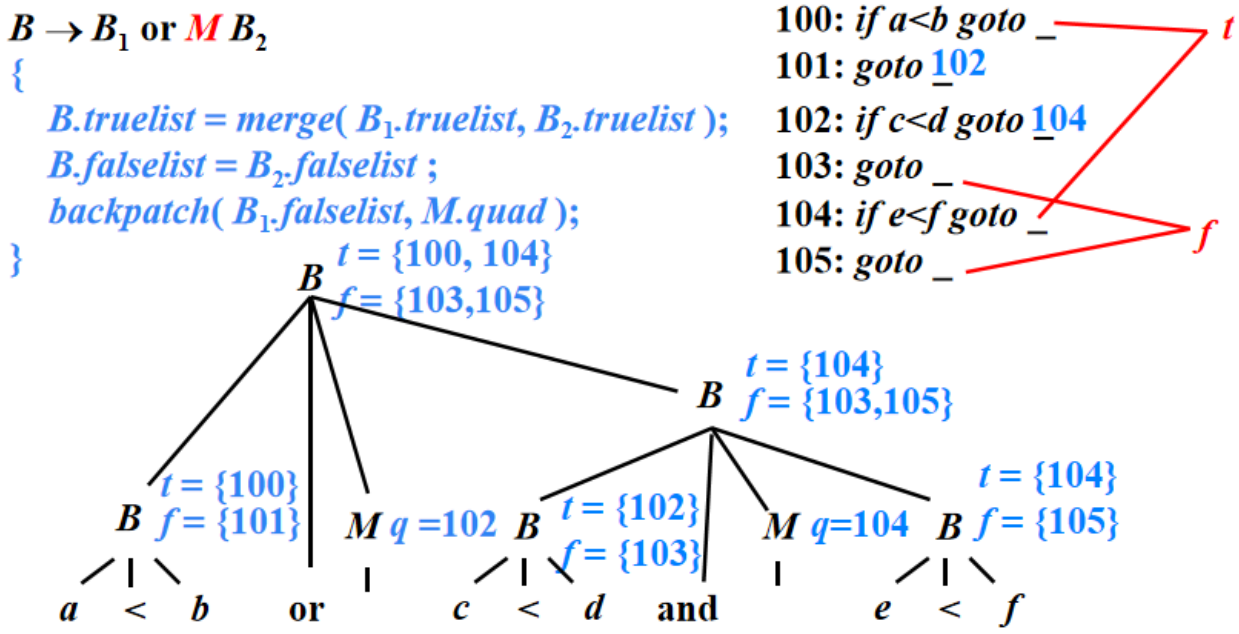
B.falselist = merge(B_1.falselist, B_2.falselist);

backpatch(B_1.truelist, M.quad);

}

100: *if a<b goto _*
101: *goto _*
102: *if c<d goto* 104
103: *goto _*
104: *if e<f goto _*
105: *goto _*





控制流语句的回填

- 文法
  - $S \rightarrow S_1 S_2$
  - $S \rightarrow \text{id} = E ; \mid L = E ;$
  - $S \rightarrow \text{if } B \text{ then } S_1$
    $\mid \text{if } B \text{ then } S_1 \text{ else } S_2$
    $\mid \text{while } B \text{ do } S_1$
- 综合属性
  - ***S.next1ist***：指向一个包含跳转指令的列表，这些指令最终获得的目标标号就是按照运行顺序紧跟在S代码之后的指令的标号

S → if B then S1

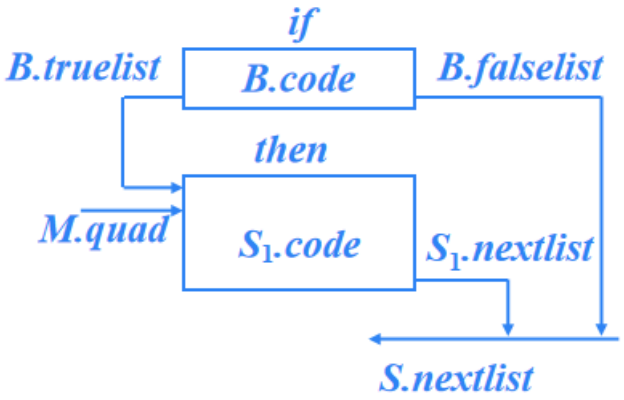
$S \rightarrow$ if B then $M\ S_1$

{

S.nextlist=merge(B.falselist, S_1.nextlist);

backpatch(B.truelist, M.quad);

}



S → if B then S1 else S2

$S \rightarrow$ if B then $M_1\ S_1\ N$ else $M_2\ S_2$

{

S.nextlist = merge(merge(S_1.nextlist, N.nextlist), S_2.nextlist);

backpatch(B.truelist, M_1.quad);

backpatch(B.falselist, M_2.quad);

}

$N \rightarrow \varepsilon$

{

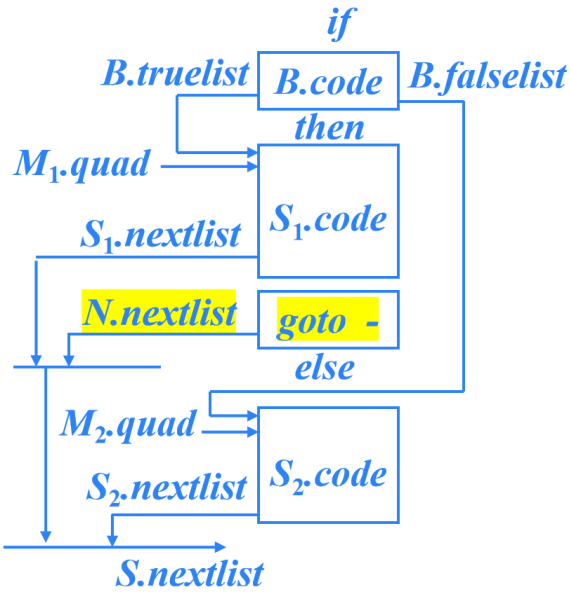
N.nextlist = makelist(nextquad);

gen('goto _');

}

$M \rightarrow \varepsilon$

{ M.quad = nextquad ; }



S → while B do S1

$S \rightarrow$ while M_1 B do M_2 S_1

{

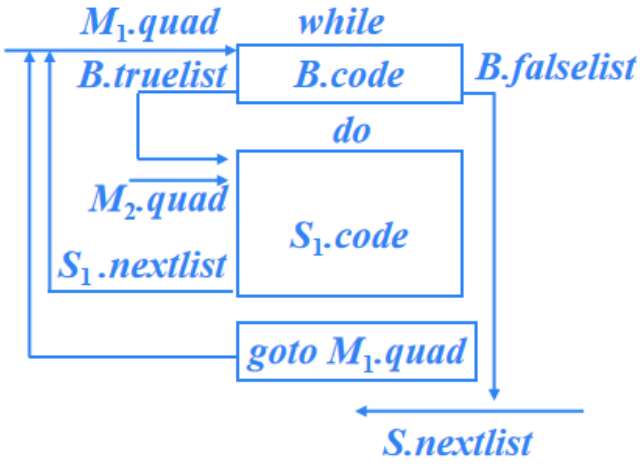
$S.nextlist = B.falselist;$

$backpatch(S_1.nextlist, M_1.quad);$

$backpatch(B.truelist, M_2.quad);$

$gen('goto'\ M_1.quad);$

}



S → S1 S2

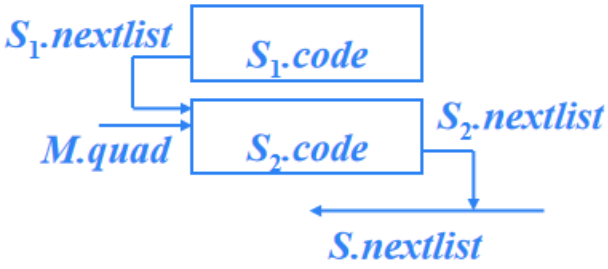
$S \rightarrow S_1\ M\ S_2$

{

$S.nextlist = S_2.nextlist;$

$backpatch(S_1.nextlist,\ M.quad);$

}



S → id = E ; | L = E ;

$S \rightarrow \mathrm{id} = E\ ; \mid L = E\ ; \{\ S.nextlist = null;\ \}$

没有跳转指令

回填技术SDT编写要点

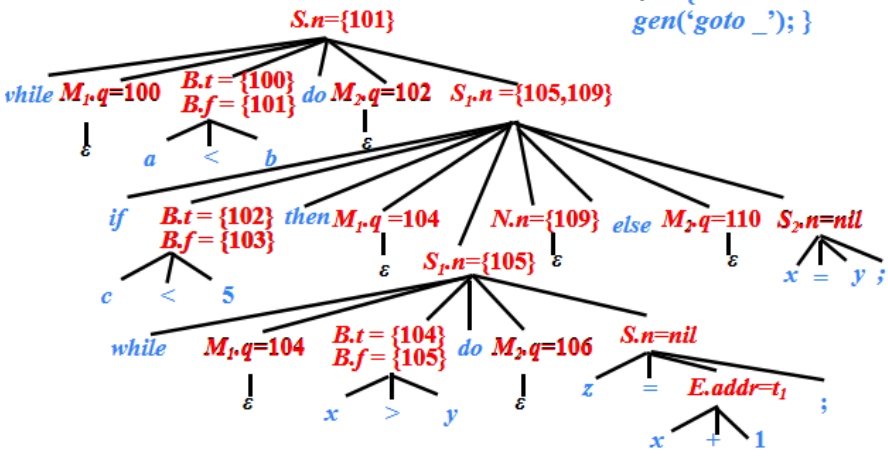
- 文法改造
  - 在list箭头指向的位置设置标记非终结符M
- 在产生式末尾的语义动作中
  - 计算综合属性
  - 调用backpatch ( )函数回填各个list
  - 生成必要的附加指令

例

$S \rightarrow$ while M_1 B do M_2 S_1
{ $S.nextlist = B.falselist$;
$backpatch(S_1.nextlist, M_1.quad)$;
$backpatch(B.truelist, M_2.quad)$;
$gen(\text{'goto'}\ M_1.quad)$; }

$S \rightarrow$ if B then M_1 S_1 N else M_2 S_2
{ $S.nextlist = merge(merge(S_1.nextlist, N.nextlist), S_2.nextlist)$;
$backpatch(B.truelist, M_1.quad)$;
$backpatch(B.falselist, M_2.quad)$; }

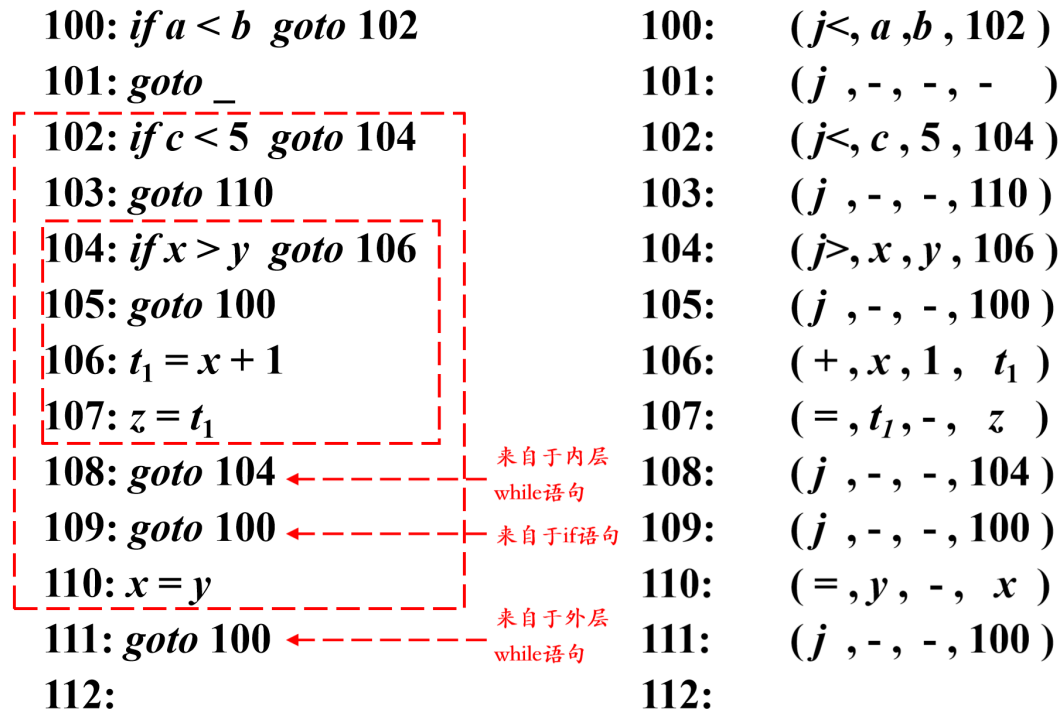
$N \rightarrow \varepsilon$ { $N.nextlist = makelist(nextquad)$;
$gen(\text{'goto}\ \_\text{'})$; }



100: *if a < b goto* 102
101: *goto* _
102: *if c < 5 goto* 104
103: *goto* 110
104: *if x > y goto* 106
105: *goto* 100
106: $t_1 = x + 1$
107: $z = t_1$
108: *goto* 104
109: *goto* 100
110: $x = y$
111: *goto* 100
112:

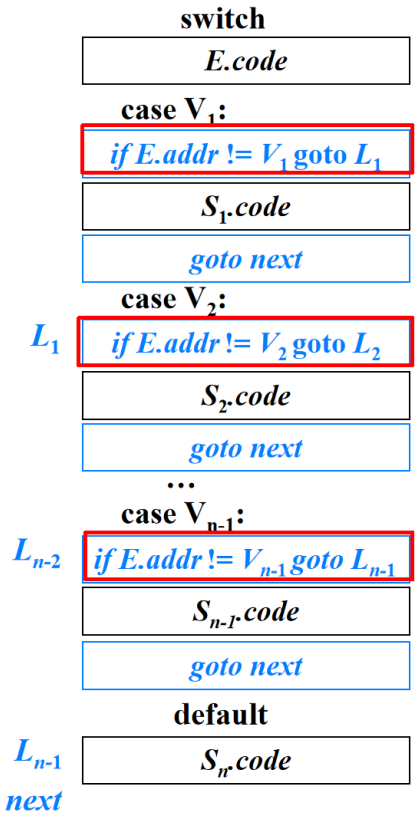
语句“*while a<b do if c<5 then while x>y do z=x+1; else x=y;*”的注释分析树

while a<b do if c<5 then while x>y do *z=x+1; else x=y;*



6.5 switch语句的翻译

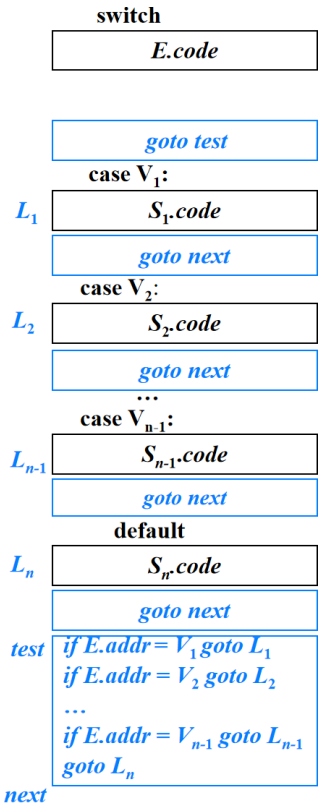
```
switch E
  begin
      case V1: S1
      case V2: S2
      . . .
      case Vn-1: Sn-1
      default: Sn
  end
```



*switch*语句的另一种翻译

```
switch E
  begin
      case V1: S1
      case V2: S2
      . . .
      case Vn-1: Sn-1
      default: Sn
  end
```

在代码生成阶段，根据分支的个数以及这些值是否在一个较小的范围内，这种条件跳转指令序列可以被翻译成最高效的*n*路分支



增加一种*case*指令

test :	*if t* = V_1 *goto* L_1
	if t = V_2 *goto* L_2
	. . .
	if t = V_{n-1} *goto* L_{n-1}
	goto L_n
next :	

test :	*case t*	V_1	L_1
	case t	V_2	L_2
	. . .		
	case t	V_{n-1}	L_{n-1}
	case t	*t*	L_n
next :			

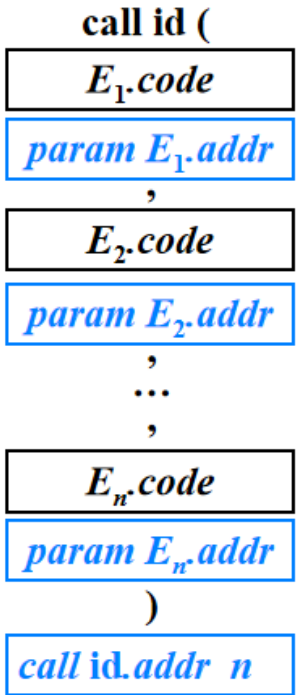
指令 *case t* V_i L_i 和 *if t* = V_i *goto* L_i 的含义相同，
但是*case*指令更加容易被最终的代码生成器探测到，
从而对这些指令进行特殊处理

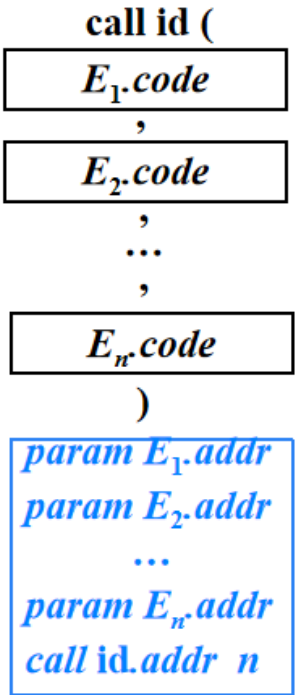
6.6 过程调用语句的翻译

- 文法
  - $S \rightarrow$ call id (*Elist*)
    Elist $\rightarrow$ *Elist*, *E*
    Elist $\rightarrow$ *E*

过程调用语句的代码结构

call id($E_1, E_2, \dots, E_n$)





- 需要一个**队列**q存放E1.addr 、E2.addr、...、 En.addr，以生成

过程调用语句的*SDD*

➢ $S \rightarrow$ call id (*Elist*)

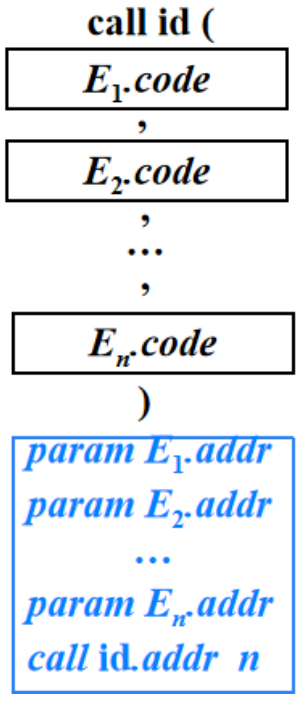
{ $n=0$;
for q中的每个t do
{ *gen*('param' *t*);
$n = n+1$;
}
gen('call' id.addr ',' *n*);
}

➢ $Elist \rightarrow E$

{ 将*q*初始化为只包含*E.addr*; }

➢ $Elist \rightarrow Elist_1, E$

{ 将*E.addr*添加到*q*的队尾; }



例：翻译以下语句$f(b*c-1, x+y, x, y)$

$$
\begin{aligned}
t_1 &= b*c \\
t_2 &= t_1 - 1 \\
t_3 &= x+y \\
&param\ t_2 \\
&param\ t_3 \\
&param\ x \\
&param\ y \\
&call\ f, 4
\end{aligned}
$$

第七章 运行存储分配

7.1 存储组织

- 一个目标程序运行所需的存储空间主要包括
  - **代码区**（算法）
  - **数据区**（数据结构）

- **存储分配策略**
  - **静态存储分配（编译时刻）**

在**编译时刻**就可以确定大小的数据对象，可以在编译时刻就为它们分配存储空间

- **动态存储分配（运行时刻）**

  在编译时仅产生各种必要的信息，而在**运行时刻**，再动态地分配数据对象的存储空间

  - **栈式存储分配**
  - **堆式存储分配**

- **运行时的内存划分**

  - 要**尽可能多**的将数据对象进行**静态分配**，因为这些对象的**地址**可以被编译到**目标代码**中

- **活动记录**

- 使用过程(或函数、方法)作为用户自定义动作的单元的语言，其编译器通常以过程为单位分配存储空间
- 过程体的每次执行称为该过程的一个活动(*activation*)
- 过程每执行一次，就为它分配一块连续存储区，用来管理过程一次执行所需的信息，这块连续存储区称为活动记录(*activation record*)

  - **活动记录的一般形式**

调用过程提供给被调用过程的参数

被调用过程返回给调用过程的值

指向调用者的活动记录

用来访问存放于其它活动记录中的非局部数据

通常包括返回地址和一些寄存器中的内容

在该过程中声明的数据

比如表达式求值过程中产生的临时变量

7.2 静态存储分配

- 在静态存储分配中，编译器为每个过程确定其活动记录在目标程序中的位置
- 这样，过程中每个名字的存储位置就确定了
- 因此，这些名字的存储地址可以被编译到目标代码中
- 过程每次执行时，它的名字都绑定到同样的存储单元

静态存储分配的限制条件

- 适合静态存储分配的语言必须满足以下条件
  - 数组上下界必须是常数
  - 不允许过程的递归调用
  - 不允许用户动态建立数据实体
- 满足这些条件的语言有BASIC和FORTRAN等

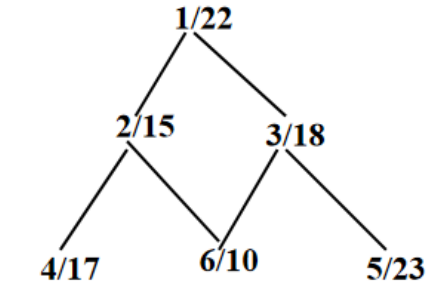
常用的静态存储分配方法

- 顺序分配法
- 层次分配法

顺序分配法

➢ 按照过程出现的先后顺序逐段分配存储空间

➢ 各过程的活动记录占用互不相交的存储空间



过程	存储区域
1	0~21
2	22~36
3	37~54
4	55~71
5	72~94
6	95~104

共需要105个存储单元

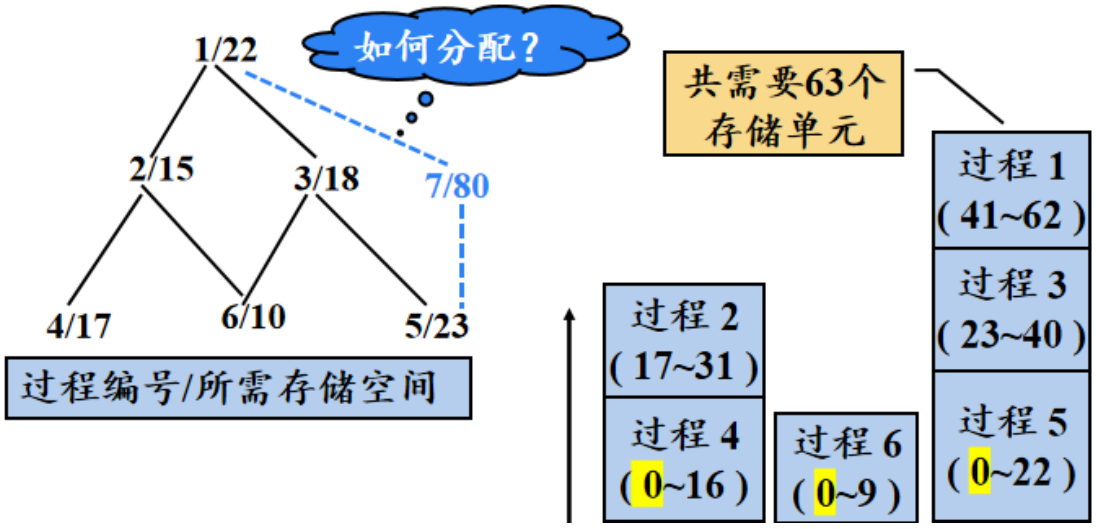
优点：处理上简单

缺点：对内存空间的使用不够经济合理

- **父节点过程会调用子节点过程**

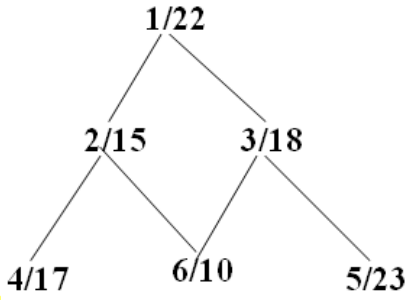
层次分配法

➢ 通过对过程间的调用关系进行分析，凡属无相互调用关系的并列过程，尽量使其局部数据共享存储空间



- **层次分配算法**

- *B*[*n*][*n*]：过程调用关系矩阵
  - *B*[*i*][*j*]=1：表示第*i*个过程调用第*j*个过程
  - *B*[*i*][*j*]=0：表示第*i*个过程不调用第*j*个过程



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	0	1	0	1
3	0	0	0	0	1	1
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0

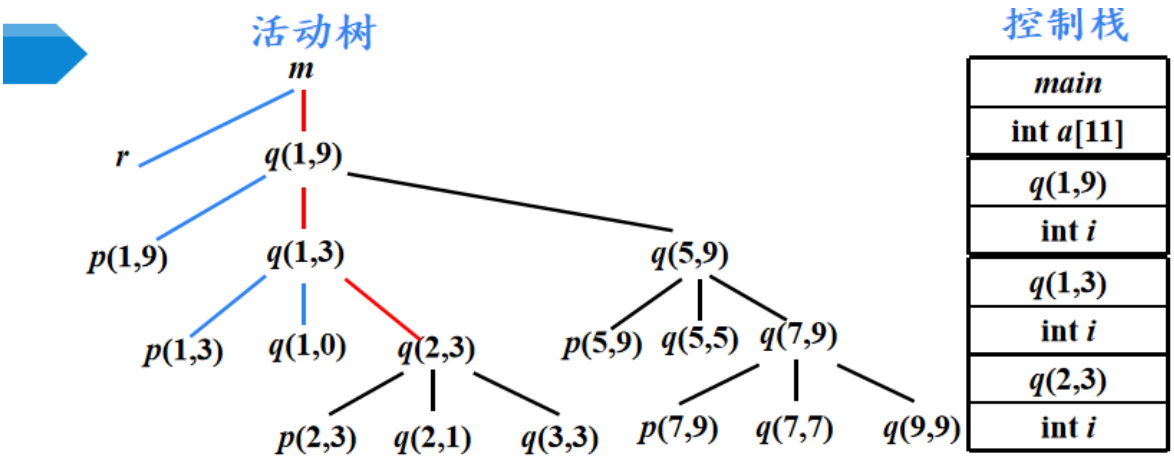
- *Units*[*n*]：过程所需内存量矩阵

7.3 栈式存储分配

- 有些语言使用过程、函数或方法作为用户自定义动作的单元，几乎所有针对这些语言的编译器都把它们的(至少一部分的)运行时刻存储以栈的形式进行管理，称为栈式存储分配
- 当一个过程被调用时，该过程的活动记录被压入栈；当过程结束时，该活动记录被弹出栈
- 这种安排不仅允许活跃时段不交叠的多个过程调用之间共享空间，而且允许以如下方式为一个过程编译代码：它的非局部变量的相对地址总是固定的，和过程调用序列无关

活动树

- 用来描述程序运行期间控制进入和离开各个活动的情况的树称为活动树
- 树中的每个结点对应于一个活动。根结点是启动程序执行的*main*过程的活动
- 在表示过程*p*的某个活动的结点上，其子结点对应于被*p*的这次活动调用的各个过程的活动。按照这些活动被调用的顺序，自左向右地显示它们。一个子结点必须在其右兄弟结点的活动开始之前结束

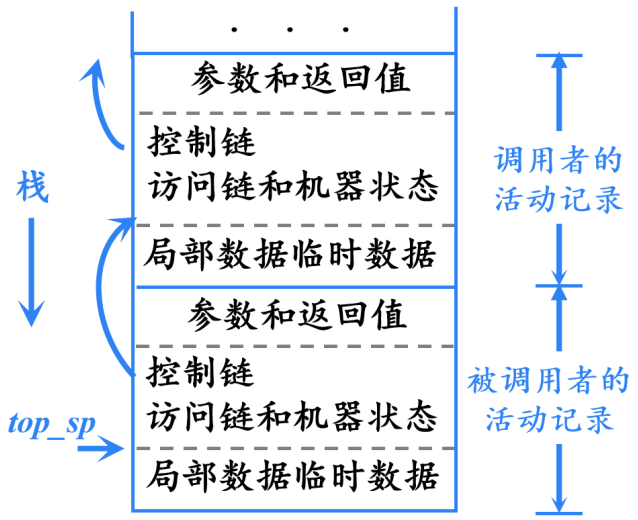


- 每个活跃的活动都有一个位于控制栈中的活动记录
- 活动树的根的活动记录位于栈底
- 程序控制所在的活动的记录位于栈顶
- 栈中全部活动记录的序列对应于在活动树中到达当前控制所在的活动结点的路径

- **当一个过程是递归的时候，常常会有该过程的多个活动记录同时出现在栈中**

设计活动记录的一些原则

- 在调用者和被调用者之间传递的值一般被放在被调用者的活动记录的开始位置，这样它们可以尽可能地靠近调用者的活动记录
- 固定长度的项被放置在中间位置
  - 控制连、访问链、机器状态字
- 在早期不知道大小的项被放置在活动记录的尾部
- 栈顶指针寄存器*top_sp*指向活动记录中局部数据开始的位置，以该位置作为基地址

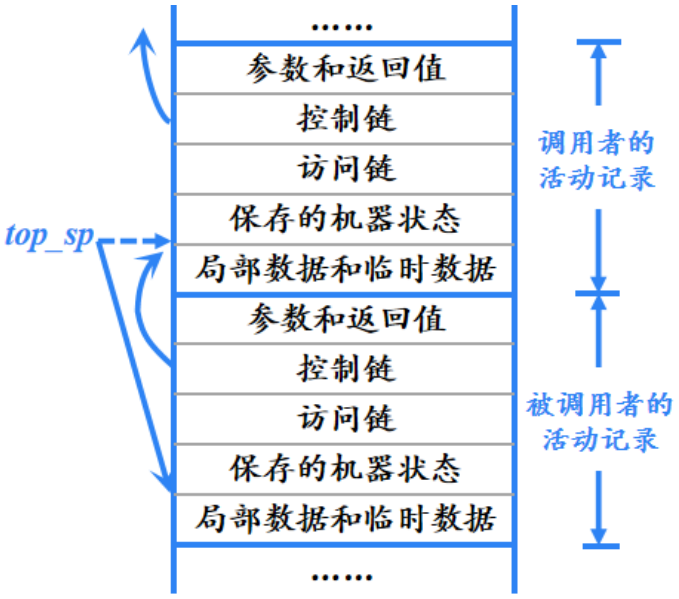


调用序列和返回序列

- 过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等
  - 调用序列
    - 实现过程调用的代码段。为一个活动记录在栈中分配空间，并在此记录的字段中填写信息
  - 返回序列
    - 恢复机器状态，使得调用过程能够在调用结束之后继续执行
  - 一个调用代码序列中的代码通常被分割到调用过程（调用者）和被调用过程（被调用者）中。返回序列也是如此

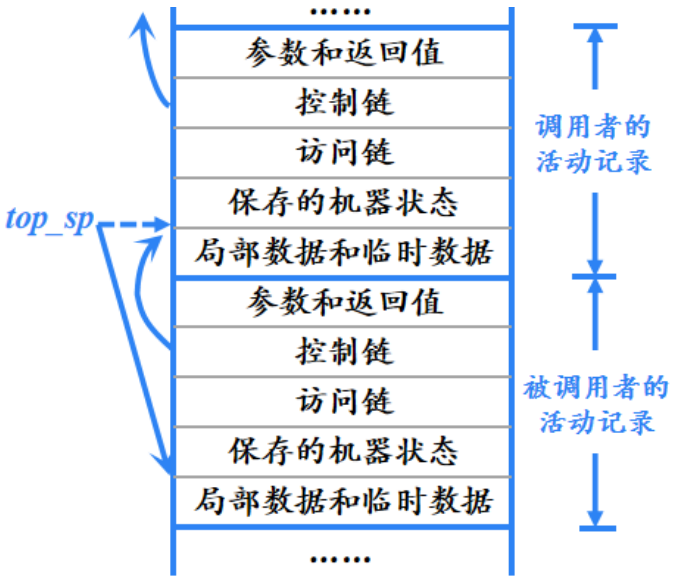
调用序列

- 调用者计算实际参数的值
- 调用者将返回地址（程序计器的值）放到被调用者的机器状态字段中。将原来的*top-sp*值放到被调用者的控制链中。然后，增加*top-sp*的值，使其指向被调用者局部数据开始的位置
- 被调用者保存寄存器值和其它状态信息
- 被调用者初始化其局部数据并开始执行

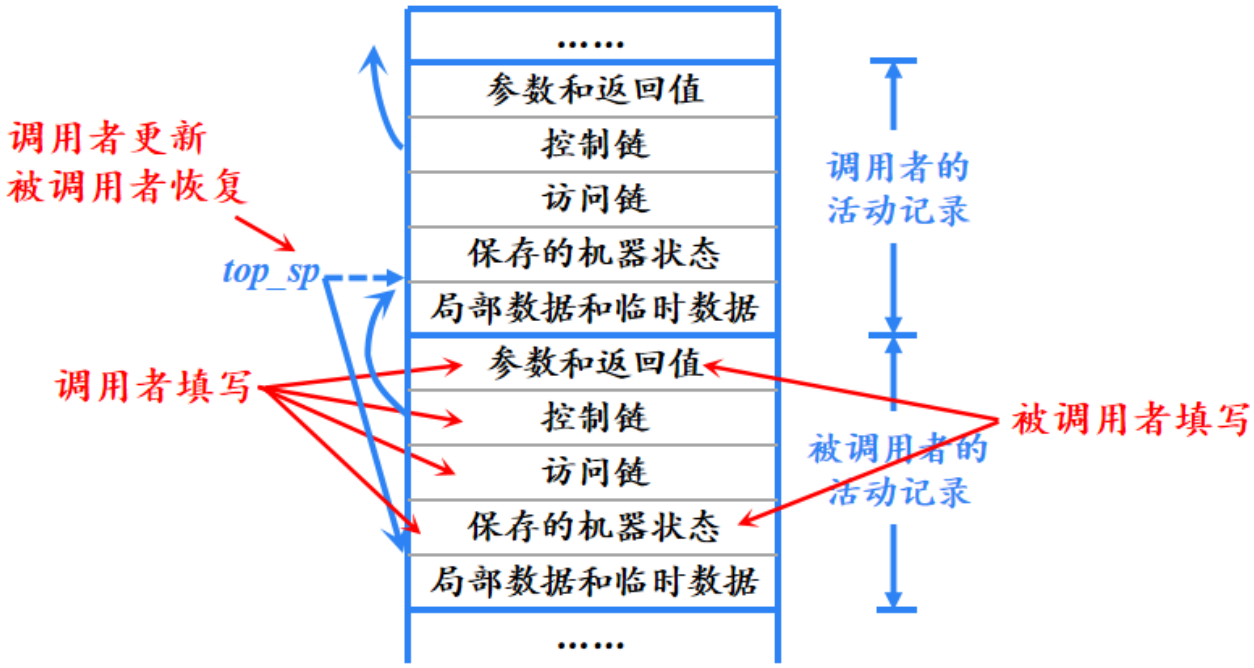


返回序列

- 被调用者将返回值放到与参数相邻的位置
- 使用机器状态字段中的信息，被调用者恢复*top-sp*和其它寄存器，然后跳转到由调用者放在机器状态字段中的返回地址
- 尽管*top-sp*已经被减小，但调用者仍然知道返回值相对于当前*top-sp*值的位置。因此，调用者可以使用那个返回值



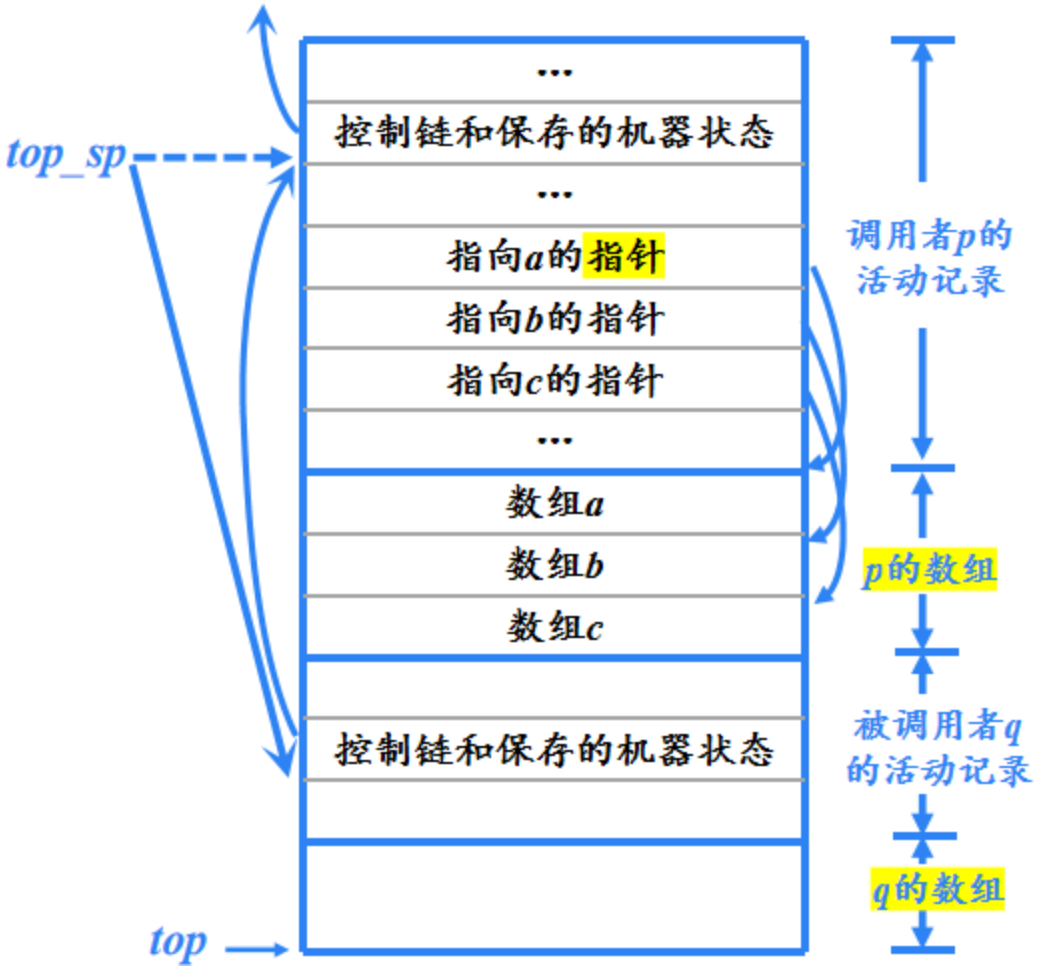
调用者和被调用者之间的任务划分



变长数据的存储分配

- 在现代程序设计语言中，在编译时刻不能确定大小的对象将被分配在堆区。但是，如果它们是过程的局部对象，也可以将它们分配在运行时刻栈中。尽量将对象放置在栈区的原因：可以避免对它们的空间进行垃圾回收，也就减少了相应的开销
- 只有一个数据对象局部于某个过程，且当此过程结束时它变得不可访问，才可以使用栈为这个对象分配空间

访问动态分配的数组



7.4 非局部数据的访问

- 一个过程除了可以使用**过程自身**声明的**局部数据**以外，还可以使用**过程外**声明的**非局部数据**
  - **非局部数据**
    - 全局数据
    - 外围过程定义的数据（支持过程嵌套声明的语言 eg.Pascal语言）
  - 语言类型

- **支持**过程嵌套声明的语言

  可以在一个过程中声明另一个过程 eg. Pascal

  一个过程除了**自身定义的局部数据**和**全局定义**的数据以外，还可以使用**外围过程中声明的对象**

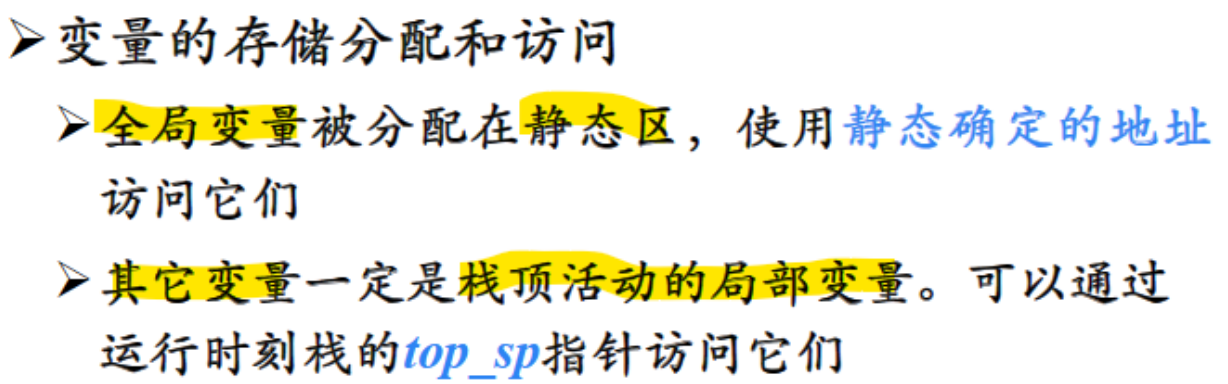
- **不支持**过程嵌套声明的语言

  不可以在一个过程中声明另一个过程 eg.C语言

  过程中使用的数据要么是自身定义的**局部数据**，要么是在所有过程之外定义的**全局数据**

- 如何**访问非局部数据**
  - **访问链（静态链）**
  - **display表（嵌套层次显示表）**

无过程嵌套声明时的数据访问



有过程嵌套声明时的数据访问

- **嵌套深度**
  - **过程的嵌套深度**
    - **不内嵌在任何其它过程**中的过程,设其嵌套深度为1
    - 如果一个过程p在一个嵌套深度为 i 的过程中定义,则设定p的嵌套深度为i+1
  - **变量的嵌套深度**

- 将变量声明所在过程的嵌套深度作为该变量的嵌套深度

例

```
program sort ( input, output );
    var a: array[0..10] of integer;
        x: integer;
    procedure readarray;
            var i: integer;
            begin … a … end {readarray} ;
    procedure exchange(i, j:integer);
            begin x=a[i];a[i]=a[j];a[j]=x; end {exchange} ;
    procedure quicksort(m, n:integer);
            var k, v : integer;
            function partition(y, z:integer):integer;
                    var i, j : integer;
                    begin … a … v … exchange(i, j) … end {partition};
            begin … a … v … partition … quicksort … end {quicksort} ;
    begin … a … readarray … quicksort … end {sort};
```

过程	嵌套深度
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

访问链

- 静态作用域规则：只要过程*b*的声明嵌套在过程*a*的声明中，过程*b*就可以访问过程*a*中声明的对象
- 可以在相互嵌套的过程的活动记录之间建立一种称为访问链(*Access link*)的指针，使得内嵌的过程可以访问外层过程中声明的对象
  - 如果过程*b*在源代码中直接嵌套在过程*a*中(*b*的嵌套深度比*a*的嵌套深度多1)，那么*b*的任何活动中的访问链都指向最近的*a*的活动

访问链的建立

建立访问链的代码属于**调用序列**的一部分

嵌套深度是在**编译阶段**通过**静态分析**就能确定的

假设嵌套深度为nx的过程x调用嵌套深度为ny的过程y (x→y)

1. nx < ny的情况(**外层调用内层**)
    - y一定是直接在x中定义的 (例如：s→q, q→p) ，因此，**ny=nx +1**
    - 在调用代码序列中增加一个步骤：在y的**访问链**中放置一个**指向x的活动记录的指针**
2. nx = ny的情况(**本层调用本层**)
    - 例如：递归调用 (q→q)
    - 被调用者的活动记录的访问链与调用者的活动记录的访问链是相同的，可以**直接复制**
3. nx > ny的情况(**内层调用外层**，如： p→e)
    - 调用者x必定嵌套在某个过程z中，而z中**直接定义**了被调用者y
    - 从x的活动记录开始，沿着访问链经过**nx - ny + 1**步就可以找到离栈顶最近的z的活动记录。 y的访问链必须指向z的这个活动记录

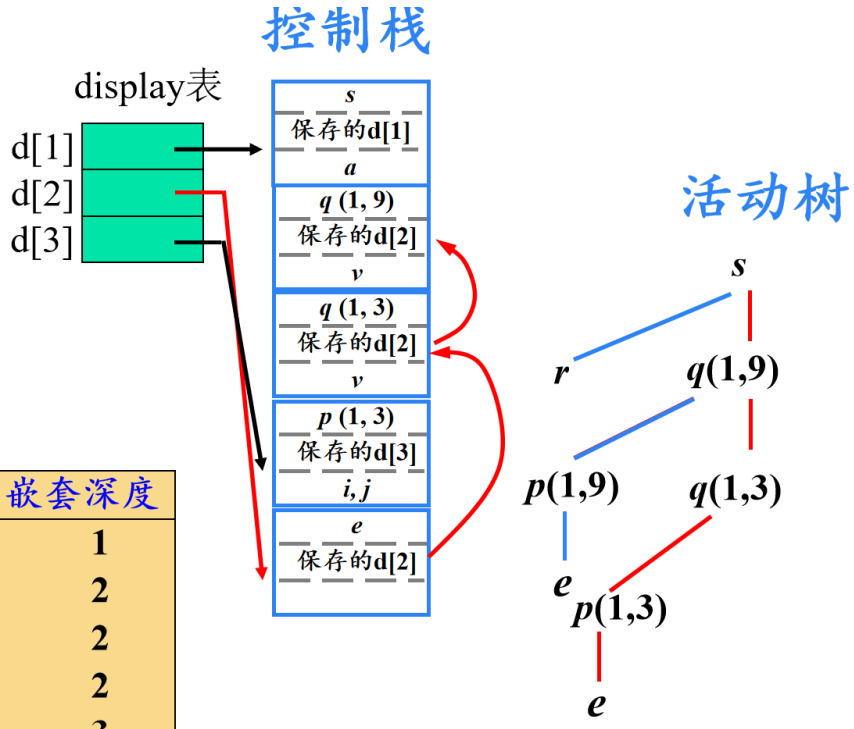
过程	嵌套深度
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

display表（嵌套层次显示表）

- 访问链方法存在的问题
  - 访问外层过程名字的效率比较低
- display表
  - 是一个指针数组d
  - 在任何时刻，$d[i]$均指向运行栈中最新建立的嵌套深度为i的过程的活动记录（在运行栈中可能存在多个嵌套深度为i的过程的活动记录）
  - 如果要访问某个嵌套深度为i的非局部名字x，只要沿着指针$d[i]$找到x所属过程的活动记录，再根据已知的偏移量就可以在活动记录中找到x

display表的维护

- 每当开始一个新活动时都要修改display表。
- 而当控制从新活动中返回时，又必须恢复display表的状态
- 如果嵌套深度为n_p的过程p被调用，并且它的活动记录不是$d[n_p]$直接指向的的活动记录，则p的活动记录要保存$d[n_p]$原来的值，同时置$d[n_p]$指向p的这个活动记录
- 当p返回且它的活动记录被从运行栈中清除时，再将$d[n_p]$恢复为调用p之前的旧值
- 如果运行栈中存在多个嵌套深度为i的过程的活动记录，则通过这些保存的$d[i]$就将它们链接在了一起，但是$d[i]$始终指向当前活跃的那个活动记录



过程	嵌套深度
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

7.5 参数传递

- 形式参数（formal parameter）
  - 在过程定义中使用的参数
- 实际参数（actual parameter）
  - 在调用过程时使用的参数
- 形参和实参相关联的几种方法
  - 传值（Call- by-Value）
  - 传地址（Call- by-Reference）
  - 传值结果（Call- by-Value-Result）
  - 传名（Call- by-Name）

7.5.1 传值（Call- by-Value）

- 把实参的值传递给相应的形参
- 实现方法
  - 调用过程把实参的值计算出来，并传递到被调用过程相应的形式单元中
  - 被调用过程中，像引用局部数据一样引用形式参数，直接访问对应的形式单元

```
…
procedure P(w,x,y,z);
begin
   y:=y*w;
   z:=z+x;
end
begin
   a:=5;
   b:=3;
   P(a+b,a-b,a,a);
   write(a);
end
```

w	8
x	2
y	5
z	5
a	5
b	3
t_1	8
t_2	2

```
…
procedure P(w,x,y,z);
begin
   y:=y*w;
   z:=z+x;
end
begin
   a:=5;
   b:=3;
   P(a+b,a-b,a,a);
   write(a);
end
```

w	8
x	2
y	40
z	7
a	5
b	3
t_1	8
t_2	2

- 将`a+b`的值计算出来`t1`，并传递给被调用过程的`w`
- **直接**修改**形参**的存储的数据

7.5.2 传地址（Call- by-Reference）

- 把实参的地址传递给相应的形参
- 实现方法
  - 调用过程把实参的地址传递到被调用过程相应的形式单元中
  - 被调用过程中，对形参的引用或赋值被处理成对形式单元的间接访问

```
…
procedure P(w,x,y,z);
begin
   y:=y*w;
   z:=z+x;
end
begin
   a:=5;
   b:=3;
   P(a+b,a-b,a,a);
   write(a);
end
```

w	t_1的地址
x	t_2的地址
y	a的地址
z	a的地址
a	5
b	3
t_1	8
t_2	2

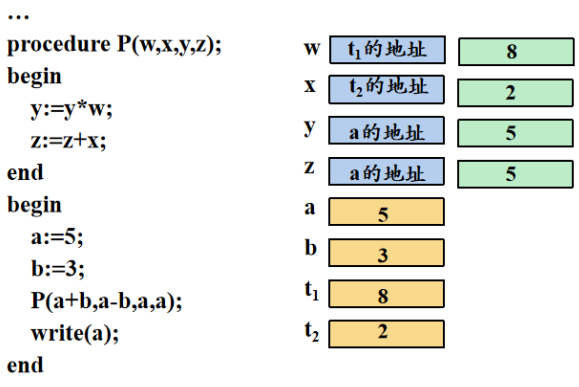
```
…
procedure P(w,x,y,z);
begin
   y:=y*w;
   z:=z+x;
end
begin
   a:=5;
   b:=3;
   P(a+b,a-b,a,a);
   write(a);
end
```

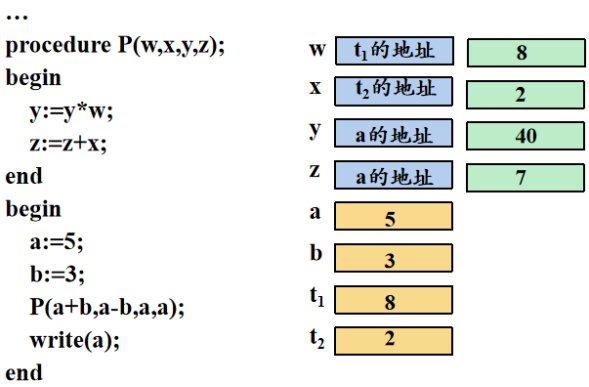
w	t_1的地址
x	t_2的地址
y	a的地址
z	a的地址
a	40
b	3
t_1	8
t_2	2

- 实际上是修改了**实参**存储的数据

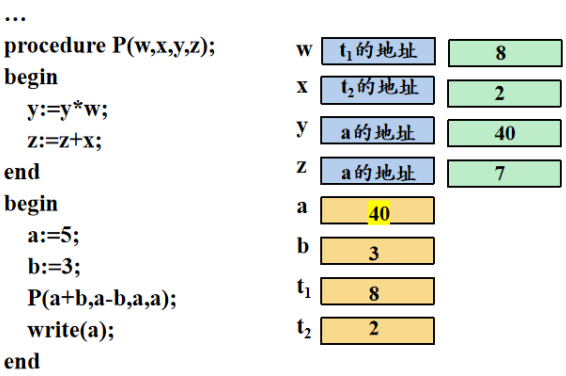
7.5.3 传值结果（Call- by-Value-Result）

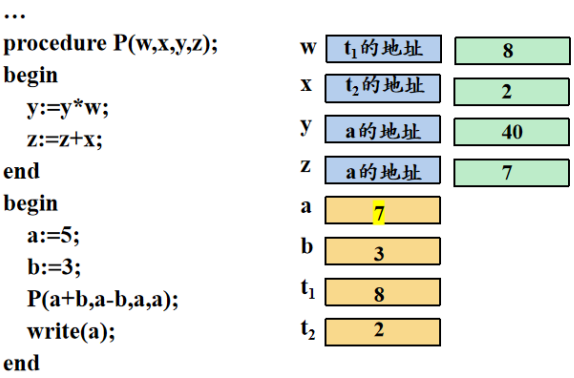
- 传地址的一种变形
- 实现方法
  - 每个形参对应两个形式单元。第一个形式单元存放实参的地址，第二个形式单元存放实参的值
  - 在过程体中，对形参的引用或赋值看作对它的第二个形式单元的直接访问
  - 过程完成返回前，把第二个单元的内容存放到第一个单元所指的实参单元中





- 在第二个形式单元中改数据





- 过程完成返回前，将第二个形式单元的内容存放到第一个形式单元所指的实参单元中

7.5.4 传名（Call- by-Name）

- 相当于把被调用过程的过程体抄到调用出现的地方，但把其中出现的形参都替换成相应的实参
- 实现方法
  - 在进入被调用过程之前不对实参预先进行计值，而是让过程体中每当使用到相应的实参时才逐次对它实行计值（或计算地址）
  - 通常把实参处理成一个子程序（称为参数子程序），每当过程体中使用到相应的实参时就调用这个子程序

```
…
procedure P(w,x,y,z);
begin
   y:=y*w;
   z:=z+x;
end
begin
   a:=5;
   b:=3;
   P(a+b,a-b,a,a);
   write(a);
end
```

```
begin
   a:=5;
   b:=3;
   a:=a*(a+b);
   a:=a+(a-b);
   write(a);
end
```

a	5
b	3

传名方式中的重命名问题

```
PROGRAM  EX
   …
   var A: integer;
   PROCEDURE  P(B: integer)
      …
   var A: integer;
   BEGIN
      A:=0;
      B:=B+1;
      A:=A+B;
   END
BEGIN
   A:=2;
   P(A);
   write(A);
END
```

```
BEGIN
   A:=2;
   A:=0;
   B:=B+1;
   A:=A+B;
   write(A);
END
```

```
BEGIN
   A:=2;
   A:=0;
   A:=A+1;
   A:=A+A;
   write(A);
END
```

```
BEGIN
   A:=2;
   TA:=0;
   A:=A+1;
   TA:=TA+A;
   write(A);
END
```

参数传递方式对比

```
...
procedure P(w,x,y,z);
begin
   y:=y*w;
   z:=z+x;
end
begin
   a:=5;
   b:=3;
   P(a+b,a-b,a,a);
   write(a);
end
```

- 传值: 5
- 传地址: 42
- 传值结果: 7
- 传名: 77

7.6 符号表

- 符号表是用于存放标识符的属性信息的数据结构
  - 种属 (Kind)
  - 类型 (Type)
  - 存储位置、长度
  - 作用域
  - 参数和返回值信息
- 符号表的作用
  - 辅助代码生成
  - 一致性检查

NAME	TYPE	KIND	VAL	ADDR
SIMPLE	整	简变		
SYMBLE	实	数组		
TABLE	字符	常数		
⋮	⋮	⋮	⋮	⋮

符号表上的主要操作

- **声明语句**的翻译（**定义性**出现）
  - **填、查**
- **可执行**语句的翻译（**使用性**出现）
  - **查**

单个过程符号表的组织

- 方法一：一张大表
  - 问题：不同种属的名字所需存放的属性信息在数量上的差异会造成符号表空间的浪费
- 方法二：多张子表（按种属分）
  - 变量表、数组表、过程表、…
  - 问题：为避免重名问题，插入或查找某个符号时需要查看所有的符号表，从而造成时间上的浪费
- 解决办法
  - 基本属性（直接存放在符号表中）+扩展属性（动态申请内存）

```
int abc;
int i;
int myarray[3][4];
```

名字	基本属性				扩展属性
	种属	类型	地址	扩展属性指针	
abc	变量	int	0	NULL	
i	变量	int	4	NULL	
myarray	数组	int	8		
…	…				

内情向量

多个过程符号表的组织

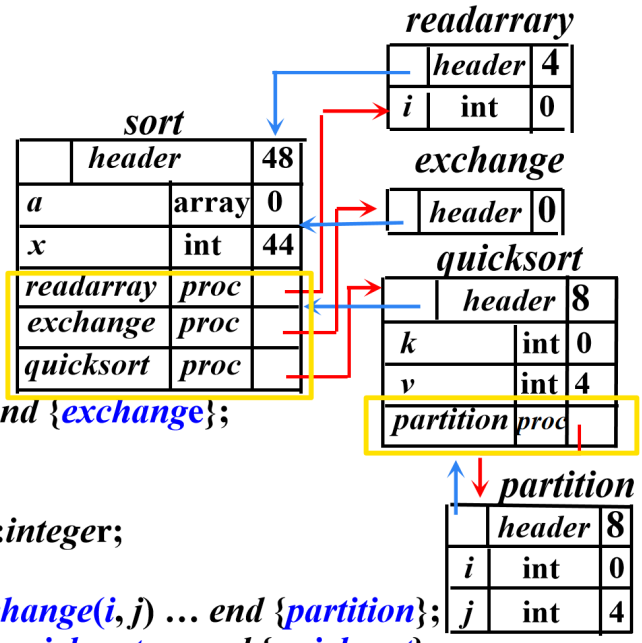
- 需要考虑的问题

  假设过程p要访问过程q中的数据对象x，
  x的地址＝q的活动记录基地址＋x在q的活动记录中的偏移地址

  - 刻画过程之间的嵌套关系（作用域信息）
  - 重名问题
- 常用的组织方式
  - 每个过程建立一个符号表，同时需要建立起这些符号表之间的联系，用来刻画过程之间的嵌套关系

例

```
program sort ( input, output );
   var a: array[0..10] of integer;
       x: integer;
   procedure readarray;
          var i: integer;
          begin … a … end {readarray};
   procedure exchange(i, j:integer);
          begin x=a[i];a[i]=a[j];a[j]=x; end {exchange};
   procedure quicksort(m, n:integer);
          var k, v : integer;
          function partition(y, z:integer):integer;
                 var i, j : integer;
                 begin … a … v … exchange(i, j) … end {partition};
          begin … a … v … partition … quicksort … end {quicksort};
   begin … a … readarray … quicksort … end {sort};
```



符号表的建立

➢ 嵌套过程声明语句的文法

$P \rightarrow D$

$D \rightarrow D\ D \mid \text{proc id ; } D\ S \mid \text{id : } T\text{ ;}$

➢ 在允许嵌套过程声明的语言中，局部于每个过程的名字可以使用第6章介绍的方法分配相对地址；当看到嵌套的过程p时，应暂时挂起对外围过程q声明语句的处理

嵌套过程声明语句的SDT

- 维护两个栈
  - tblptr（存放指向各个符号表的指针）
  - offset（跟踪各个符号表中已经分配的相对地址的值）

$P \rightarrow M\ D$ { *addwidth(top(tblptr), top(offset));*
pop(tblptr);
pop(offset); }

$M \rightarrow \varepsilon$ { *t = mktable(nil);*
push(t, tblptr);
push(0, offset); }

mktable(previous)
创建一个新的符号表，并返回指向新表的指针。参数*previous*指向先前创建的符号表(外围过程的符号表)

$D \rightarrow D_1\ D_2$

$D_p \rightarrow \text{proc id ; } N\ D_1\ S$ { *t = top(tblptr);*
addwidth(t, top(offset));
pop(tblptr);
pop(offset);
enterproc(top(tblptr), id.lexeme, t); }

addwidth(table, width)
将*table*指向的符号表中所有表项的宽度之和*width*记录在符号表的表头中

enterproc(table, name, newtable)
在*table*指向的符号表中为过程*name*建立一条记录，*newtable*指向过程*name*的符号表

$D_v \rightarrow \text{id: } T\text{ ;}$ { *enter(top(tblptr), id.lexeme, T.type, top(offset));*
top(offset) = top(offset) + T.width; }

$N \rightarrow \varepsilon$ { *t = mktable(top(tblptr));*
push(t, tblptr); push(0, offset); }

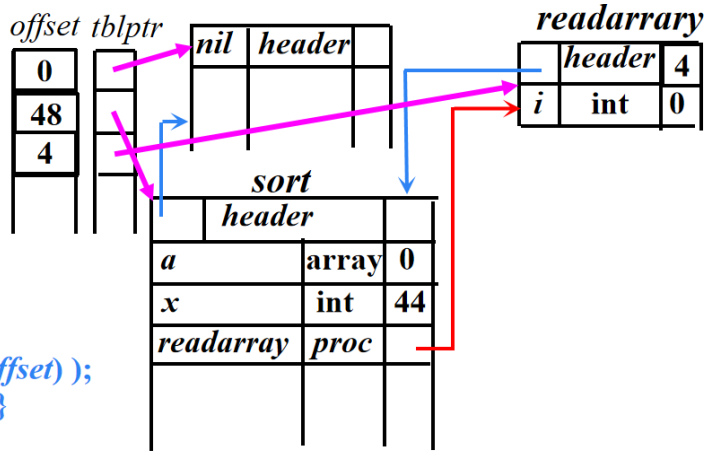
enter(table, name, type, offset)
在*table*指向的符号表中为名字*name*建立一个新表项

2. 符号表构造过程

1. **程序开始（P → M D）**
    - 调用 `M → ε` 生成一个新的符号表，并将其压入 `tblptr` 栈，同时将偏移量 `0` 压入 `offset` 栈。
2. **变量声明和过程声明（D → D1 D2）**
    - 递归地处理每个声明。
3. **变量声明（Dv → id: T;）**
    - 对于每个变量声明，将变量名和类型加入当前符号表，并更新偏移量。例如，`var a: int[11]` 和 `x: int` 会加入 `sort` 的符号表。
4. **过程声明（Dp → proc id N D1 S）**
    - 对于每个过程声明，创建一个新的符号表，并处理其局部声明和语句块。例如，`proc readarray` 和 `proc quicksort` 都会有自己的符号表。
5. **嵌套过程和函数（如 partition）**
    - 处理嵌套的过程或函数声明时，会递归地创建和管理符号表。例如，`func partition` 会有自己的符号表。

例

```
program sort;
    var a:int[11]; x:int;
    proc readarray;
        var i:int; {…}
    proc exchange; {…}
    proc quicksort;
        var k, v:int;
        func partition;
            var i, j:int; {…}
        {…}
    {…}
```



$P \rightarrow M\ D$ { *addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset); }*

$M \rightarrow \varepsilon$ { *t = mktable(nil); push(t, tblptr); push(0, offset); }*

$D \rightarrow D_1\ D_2$

$D_p \rightarrow$ proc id ; $N\ D_1\ S$ { *t = top(tblptr); addwidth(t, top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), id.lexeme, t); }*

$D_v \rightarrow$ id: T ; { *enter(top(tblptr), id.lexeme, T.type, top(offset)); top(offset) = top(offset) + T.width; }*

$N \rightarrow \varepsilon$ { *t = mktable(top(tblptr)); push(t, tblptr); push(0, offset); }*

P94

标识符的基本处理方法

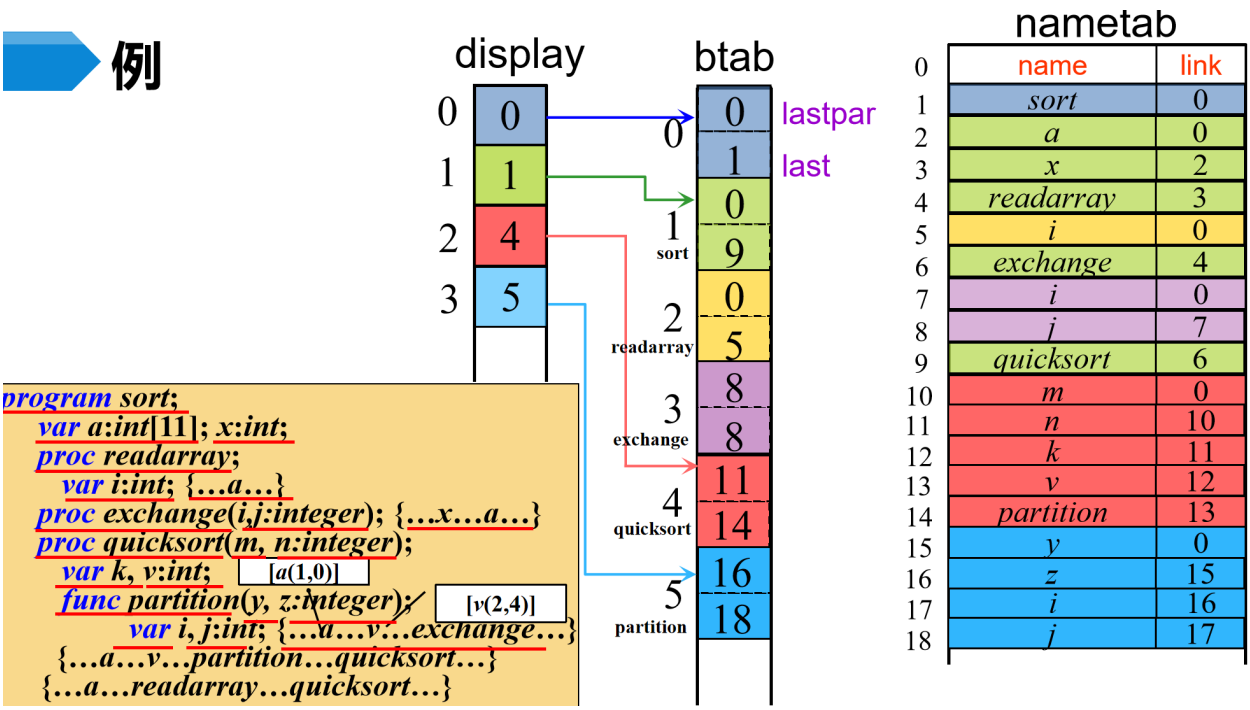
- 当在某一层的声明语句中识别出一个标识符(id的定义性出现)时，以此标识符查相应于本层的符号表
  - 如果查到，则报错并发出诊断信息“id重复声明”
  - 否则，在符号表中加入新登记项，将标识符及有关信息填入
- 当在可执行语句部分扫视到标识符时(id的应用性出现)
  - 首先在该层符号表中查找该id，如果找不到，则到直接外层符号表中去查，如此等等，一旦找到，则在表中取出有关信息并作相应处理 将层号作为中间代码中地址的一部分
  - 如果查遍所有外层符号表均未找到该id，则报错并发出诊断信息“id未声明”

符号表的另一种组织方式

- 将所有块的符号表放在一个大数组中，然后再引入一个块表来描述各块的符号表在大数组中的位置及其相互关系
  - 一个过程可以看作是一个块

- **display表**：记录下各块所在的层号，沿着该表可以找到当前正在分析的块的各个外层
- **lastpar**指向本过程体中**最后一个形参**在nametab中的位置
- **last**指向本过程体中**最后一个名字**在nametab中的位置
- **link**指向**同一过程体**中定义的**上一个名字**在nametab中的位置，每个过程体在nametab中登记的**第一个名字的link为0**

例



nametab

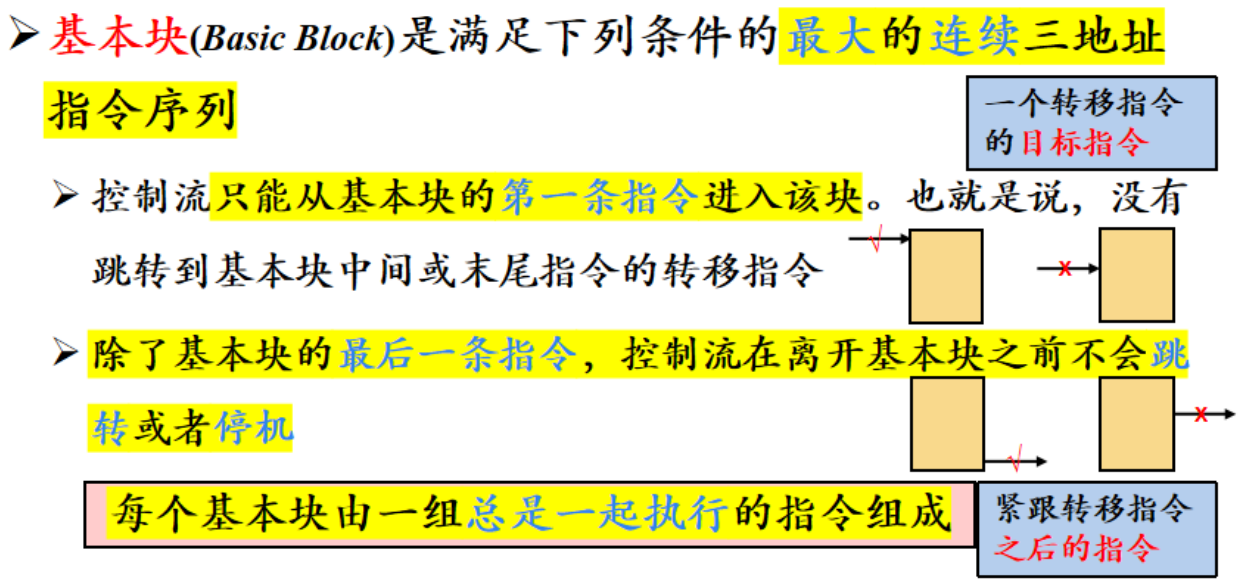
	name	link
1	sort	0
2	a	0
3	x	2
4	readarray	3
5	i	0
6	exchange	4
7	i	0
8	j	7
9	quicksort	6
10	m	0
11	n	10
12	k	11
13	v	12
14	partition	13
15	y	0
16	z	15
17	i	16
18	j	17

P105

第八章 代码优化

8.1 流图

基本块



- 基本块中的指令要么全执行，要么全不执行

基本块划分算法

- 输入：
  - 三地址指令序列
- 输出：
  - 输入序列对应的基本块列表，其中每个指令恰好被分配给一个基本块
- 方法：
  - 首先，确定指令序列中哪些指令是首指令(*leaders*)，即某个基本块的第一个指令
    1. 指令序列的第一个三地址指令是一个首指令
    2. 任意一个条件或无条件转移指令的目标指令是一个首指令
    3. 紧跟在一个条件或无条件转移指令之后的指令是一个首指令
  - 然后，每个首指令对应的基本块包括了从它自己开始，直到下一个首指令(不含)或者指令序列结尾之间的所有指令

例

```
i = m –1; j = n; v = a[n];
while (1) {
    do i = i +1; while(a[i]<v);
    do j = j –1;while (a[j]>v);
    if (i >= j) break;
    x=a[i]; a[i]=a[j]; a[j]=x;
}
x=a[i]; a[i]=a[n]; a[n]=x;
```

(1) $i = m - 1$
(2) $j = n$
(3) $t_1 = 4 * n$
(4) $v = a[t_1]$
(5) $i = i + 1$
(6) $t_2 = 4 * i$
(7) $t_3 = a[t_2]$
(8) if $t_3 < v$ goto(5)
(9) $j = j - 1$
(10) $t_4 = 4 * j$
(11) $t_5 = a[t_4]$
(12) if $t_5 > v$ goto(9)
(13) if $i >= j$ goto(23)
(14) $t_6 = 4 * i$
(15) $x = a[t_6]$
(16) $t_7 = 4 * i$
(17) $t_8 = 4 * j$
(18) $t_9 = a[t_8]$
(19) $a[t_7] = t_9$
(20) $t_{10} = 4 * j$
(21) $a[t_{10}] = x$
(22) goto (5)
(23) $t_{11} = 4 * i$
(24) $x = a[t_{11}]$
(25) $t_{12} = 4 * i$
(26) $t_{13} = 4 * n$
(27) $t_{14} = a[t_{13}]$
(28) $a[t_{12}] = t_{14}$
(29) $t_{15} = 4 * n$
(30) $a[t_{15}] = x$

B_1
(1) $i = m - 1$
(2) $j = n$
(3) $t_1 = 4 * n$
(4) $v = a[t_1]$

B_2
(5) $i = i + 1$
(6) $t_2 = 4 * i$
(7) $t_3 = a[t_2]$
(8) if $t_3 < v$ goto(5)

B_3
(9) $j = j - 1$
(10) $t_4 = 4 * j$
(11) $t_5 = a[t_4]$
(12) if $t_5 > v$ goto(9)

B_4
(13) if $i >= j$ goto(23)
(14) $t_6 = 4 * i$
(15) $x = a[t_6]$

B_5
(16) $t_7 = 4 * i$
(17) $t_8 = 4 * j$
(18) $t_9 = a[t_8]$
(19) $a[t_7] = t_9$
(20) $t_{10} = 4 * j$
(21) $a[t_{10}] = x$
(22) goto (5)

B_6
(23) $t_{11} = 4 * i$
(24) $x = a[t_{11}]$
(25) $t_{12} = 4 * i$
(26) $t_{13} = 4 * n$
(27) $t_{14} = a[t_{13}]$
(28) $a[t_{12}] = t_{14}$
(29) $t_{15} = 4 * n$
(30) $a[t_{15}] = x$

流图

- 流图的每个结点是一个基本块
- 从基本块B到基本块C之间有一条边当且仅当基本块C的第一条指令可能紧跟在B的最后一条指令之后执行

> 此时称B是C的前驱(predecessor)，C是B的后继(successor)

- 有两种方式可以确认这样的边：
  - 存在一条从B的结尾跳转到C的开头的条件或无条件跳转指令
  - 按照原来的三地址指令序列中的顺序，C紧跟在B之后，且B的结尾不存在无条件跳转指令

- **注：**按照原来的三地址指令序列中的顺序，C紧跟在B之后，**且B结尾如果是条件跳转指令是可以确定一条B→C的边的**

例

B_1
(1) $i = m - 1$
(2) $j = n$
(3) $t_1 = 4 * n$
(4) $v = a[t_1]$

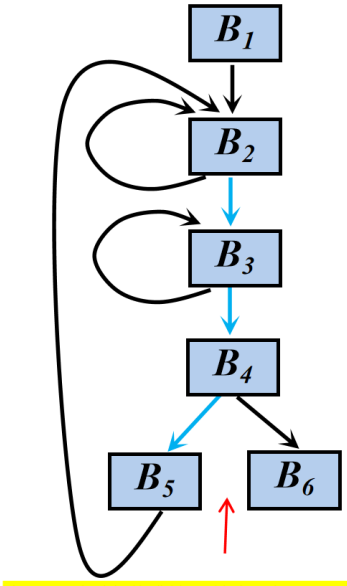
B_2
(5) $i = i + 1$
(6) $t_2 = 4 * i$
(7) $t_3 = a[t_2]$
(8) if $t_3 < v$ goto(5)

B_3
(9) $j = j - 1$
(10) $t_4 = 4 * j$
(11) $t_5 = a[t_4]$
(12) if $t_5 > v$ goto(9)

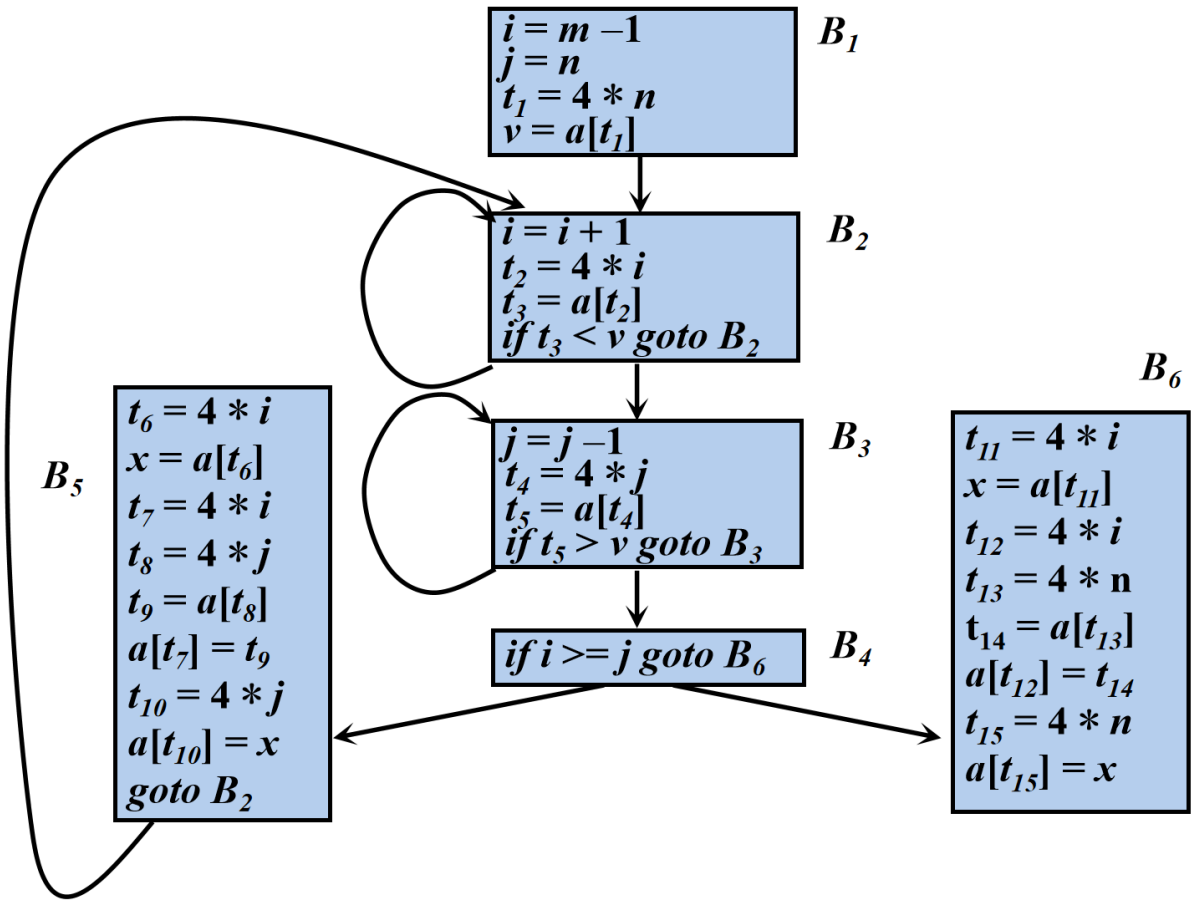
B_4
(13) if $i >= j$ goto(23)
(14) $t_6 = 4 * i$
(15) $x = a[t_6]$

B_5
(16) $t_7 = 4 * i$
(17) $t_8 = 4 * j$
(18) $t_9 = a[t_8]$
(19) $a[t_7] = t_9$
(20) $t_{10} = 4 * j$
(21) $a[t_{10}] = x$
(22) goto (5)

B_6
(23) $t_{11} = 4 * i$
(24) $x = a[t_{11}]$
(25) $t_{12} = 4 * i$
(26) $t_{13} = 4 * n$
(27) $t_{14} = a[t_{13}]$
(28) $a[t_{12}] = t_{14}$
(29) $t_{15} = 4 * n$
(30) $a[t_{15}] = x$



由于22是无条件跳转，所以没有B5->B6的边



8.2 优化的分类

- **机器无关**优化
  - 针对**中间**代码
- **机器相关**优化
  - 针对**目标**代码

- **局部**代码优化
  - **单个基本块**范围内的优化

- **全局**代码优化
  - 面向**多个基本块**的优化

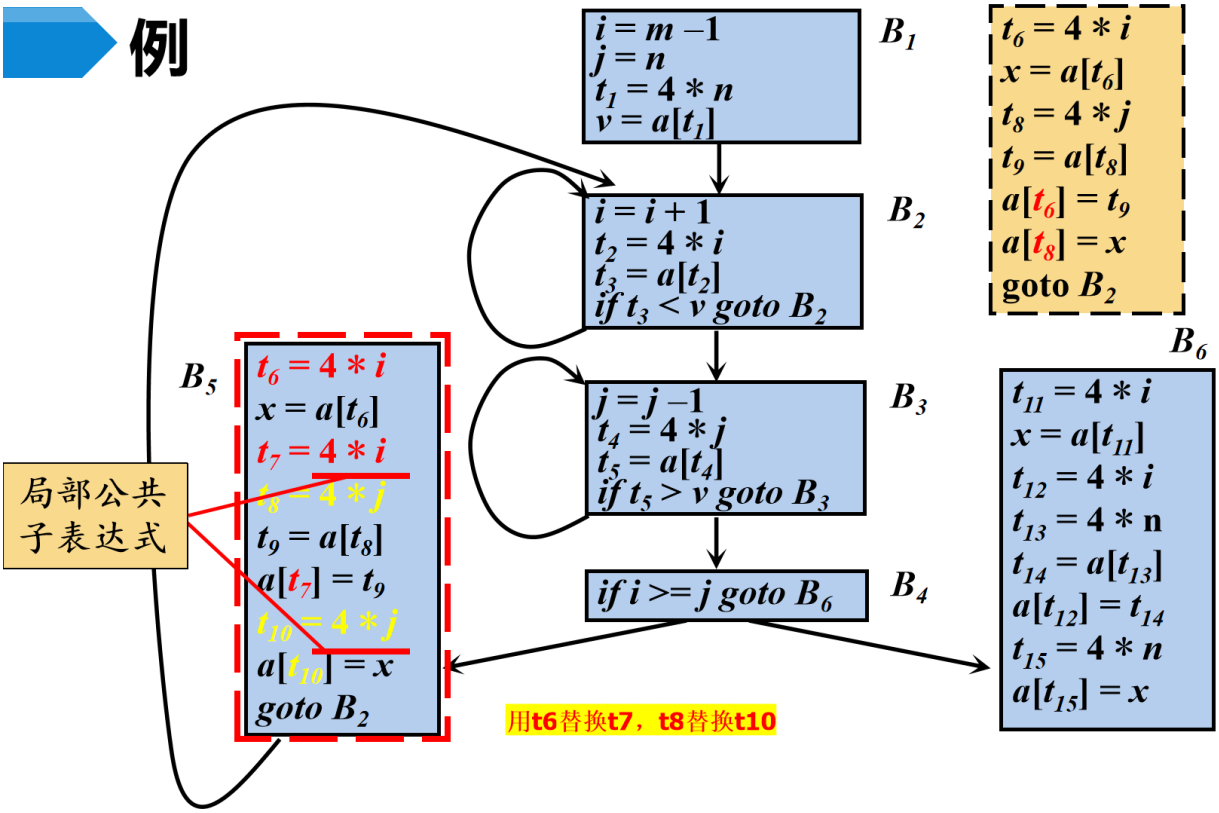
- **常用的优化方法**
  - 删除公共子表达式
  - 删除无用代码
  - 代码移动
  - 强度削弱
  - 删除归纳变量

① 删除公共子表达式

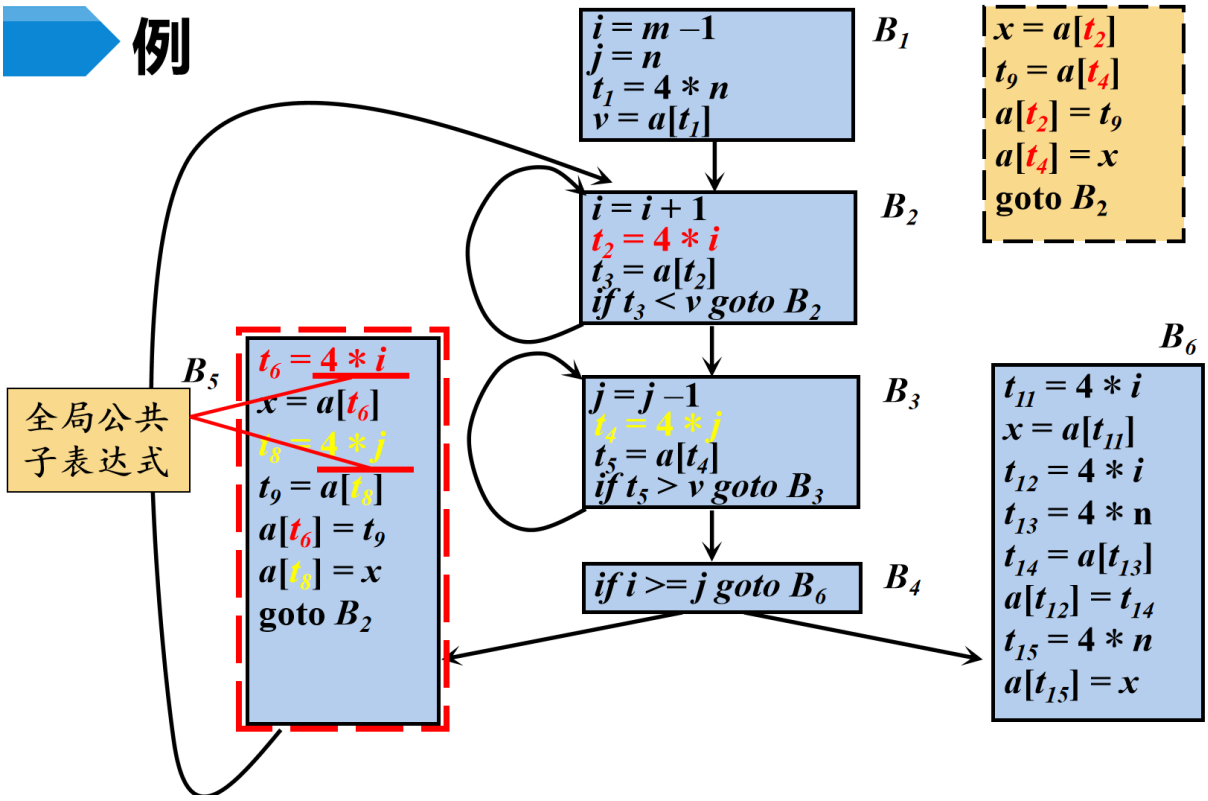
➢公共子表达式

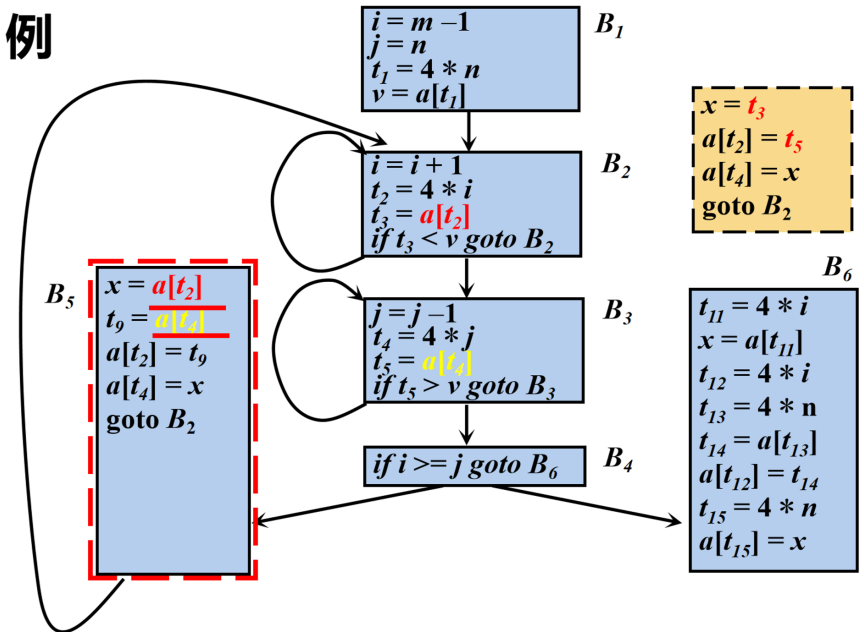
➢如果表达式x op y先前已被计算过，并且从先前的计算到现在，x op y中变量的值没有改变，那么x op y的这次出现就称为公共子表达式(*common subexpression*)

例

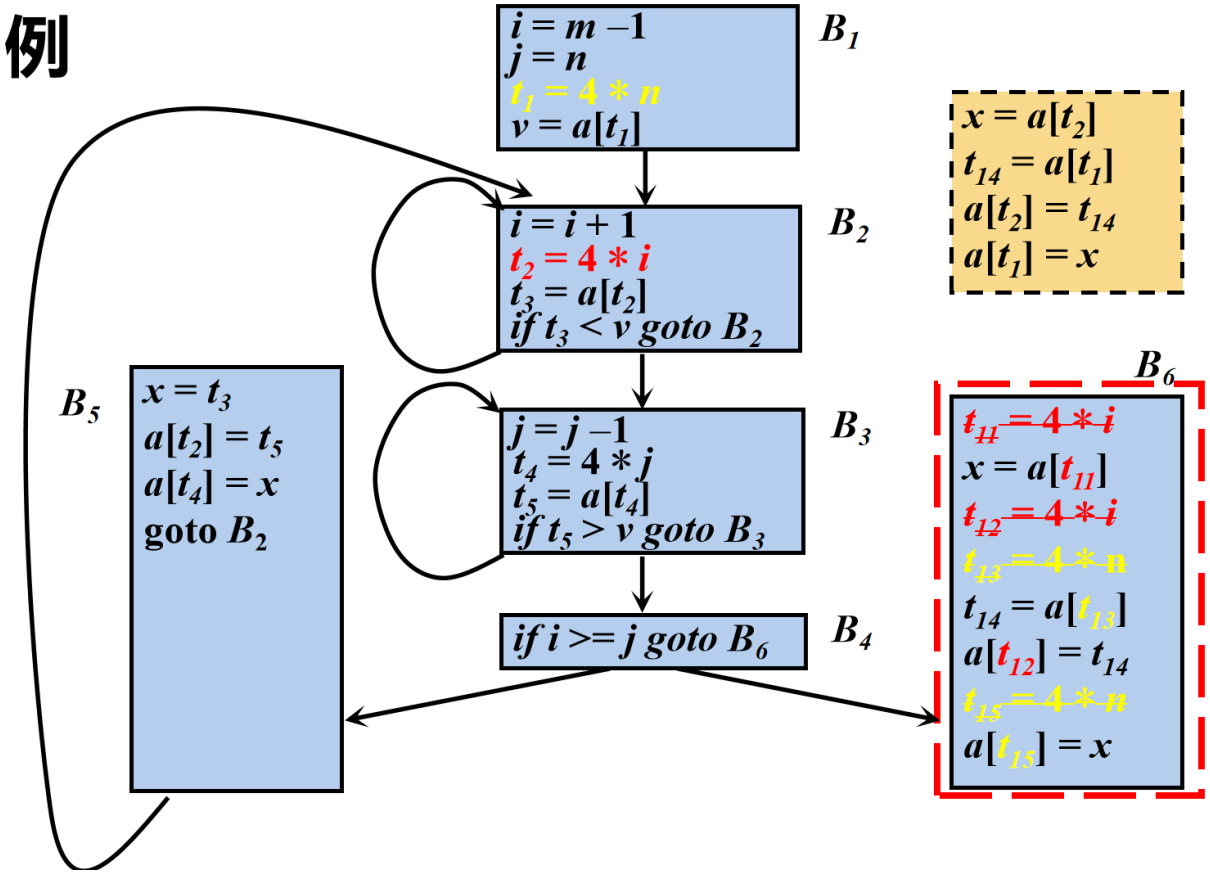

$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$
B_1
$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2
B_2
$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3
B_3
if $i >= j$ goto B_6
B_4
$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2
B_5
局部公共子表达式
$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2
B_6
$t_{11} = 4 * i$
$x = a[t_{11}]$
$t_{12} = 4 * i$
$t_{13} = 4 * n$
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4 * n$
$a[t_{15}] = x$
用t6替换t7，t8替换t10

例

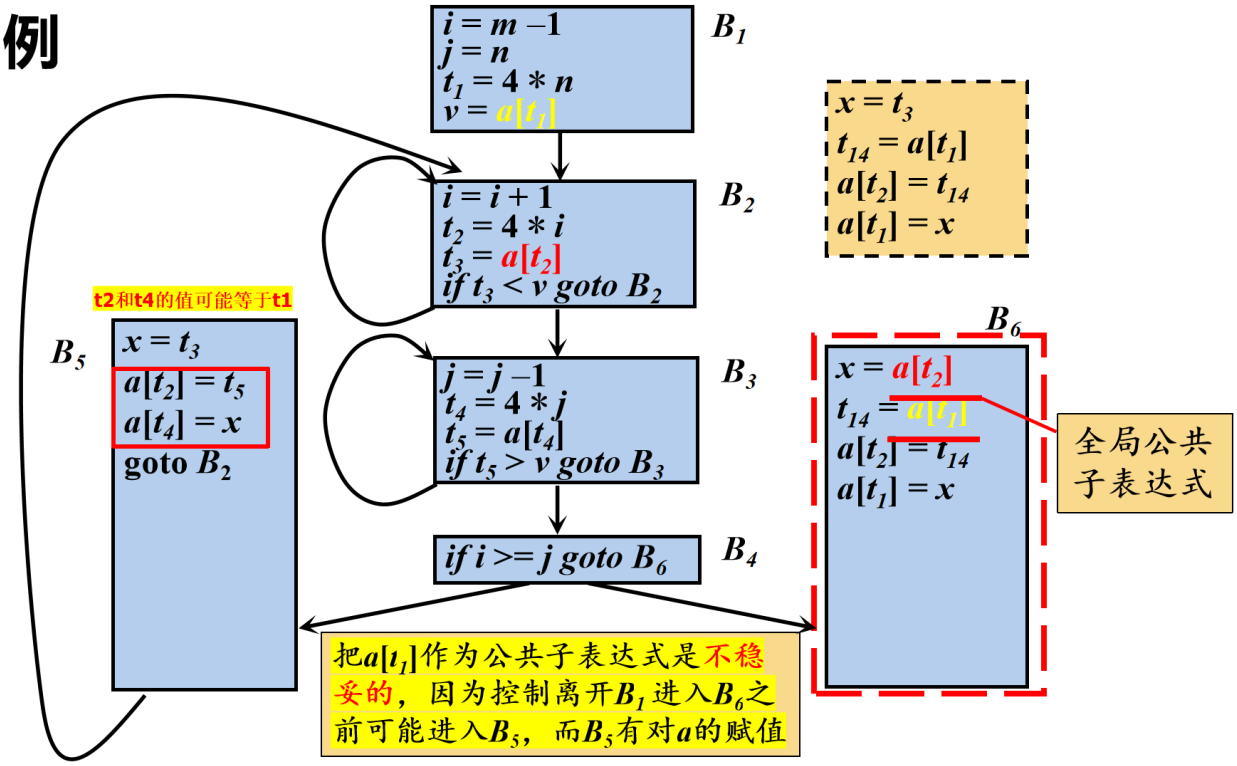

$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$
B_1
$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2
B_2
$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3
B_3
if $i >= j$ goto B_6
B_4
$x = a[t_2]$
$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B_2
B_5
全局公共子表达式
$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2
B_6
$t_{11} = 4 * i$
$x = a[t_{11}]$
$t_{12} = 4 * i$
$t_{13} = 4 * n$
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4 * n$
$a[t_{15}] = x$



- t9只是个临时变量，可直接删除



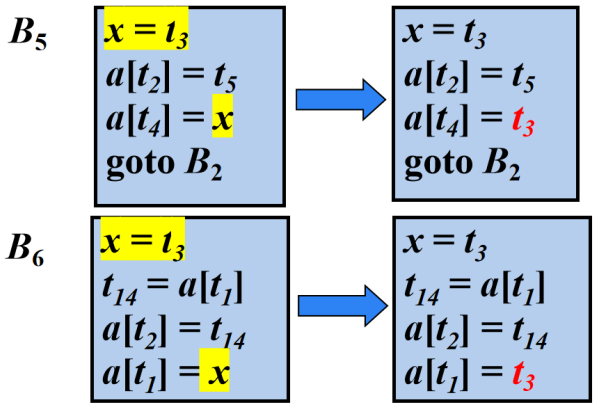
例



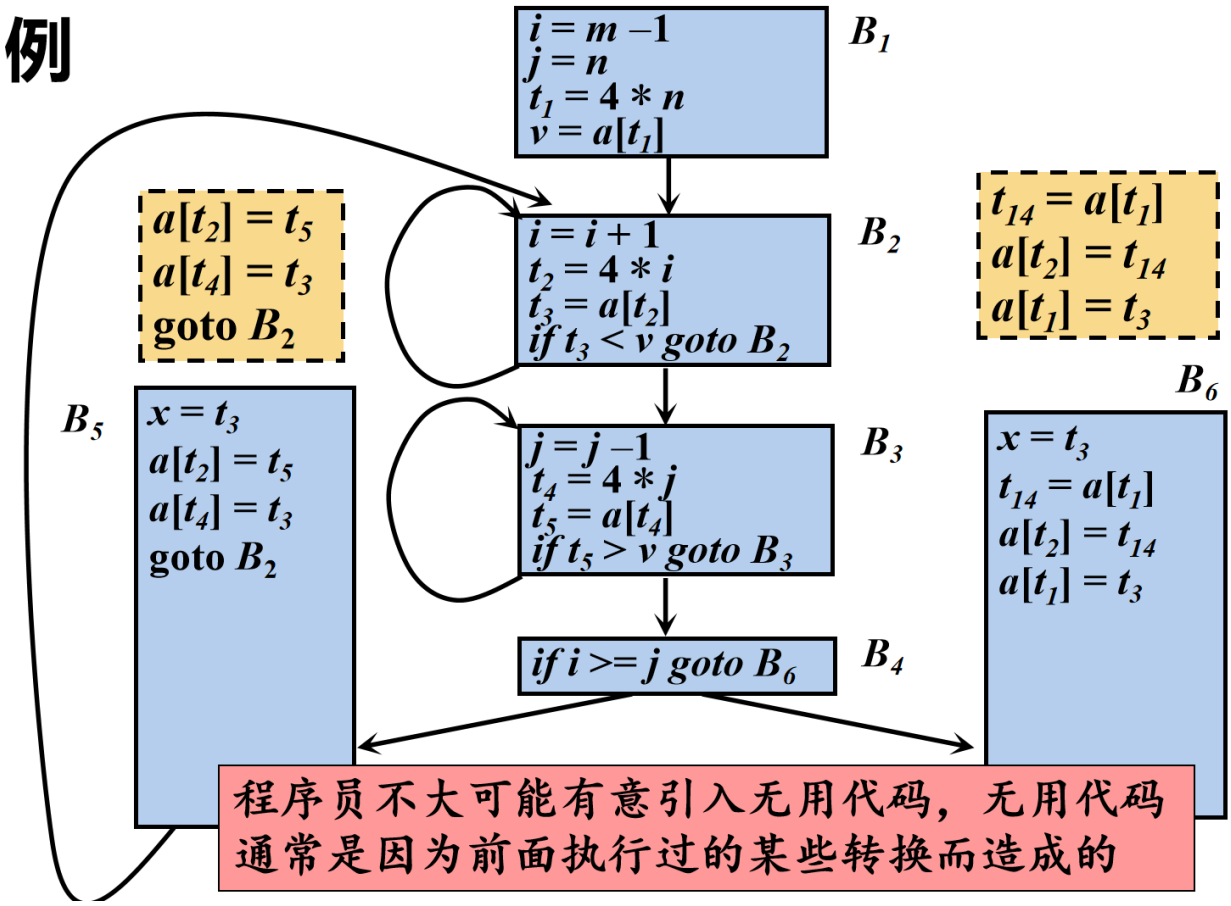
- **关键问题：**
  如何自动识别公共子表达式

② 删除无用代码

- 复制传播
  - 常用的公共子表达式消除算法和其它一些优化算法会引入一些复制语句(形如$x=y$的赋值语句)
  - 复制传播：在复制语句$x=y$之后尽可能地用y代替x
    - 例

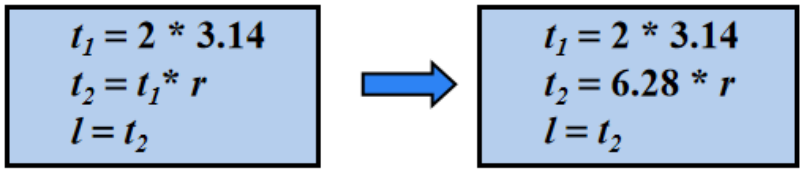


    - 复制传播给删除无用代码带来机会
- 无用代码(死代码*Dead-Code*)：其计算结果永远不会被使用的语句



常量合并(Constant Folding)

- 如果在编译时刻推导出一个表达式的值是常量，就可以使用该常量来替代这个表达式。该技术被称为常量合并（或“常量传播”）
- 例： $l = 2*3.14*r$



常量合并也能给删除无用代码带来机会

③ 代码移动

➢ 这个转换处理的是那些不管循环执行多少次都得到相同结果的表达式(即循环不变计算，*loop-invariant computation*)，在进入循环之前就对它们求值

例

原始程序

```
for( n=10; n<360; n++ )
{ S=1/360*pi*r*r*n;
  printf( "Area is %f ", S );
}
```

优化后程序

```
C= 1/360*pi*r*r;
for( n=10; n<360; n++ )
{ S=C*n;
  printf( "Area is %f ", S );
}
```

循环不变计算

```
(1) n = 10
(2) if n>=360 goto(21)
(3) goto (7)
(4) t1 = n + 1
(5) n = t1
(6) goto (2)
(7) t2 = 1 / 360
(8) t3 = t2 * pi
(9) t4 = t3 * r
(10) t5 = t4 * r
(11) t6 = t5 * n
(12) S = t6
      …
(20) goto (4)
(21)
```

循环不变计算的相对性

- 对于多重嵌套的循环，循环不变计算是相对于某个循环而言的。可能对于更加外层的循环，它就不是循环不变计算
  - 例：

```
for(i = 1; i<10; i++)
    for( n=1; n<360/(5*i); n++ )
    {  S=(5*i)/360*pi*r*r*n; ... }
```

---

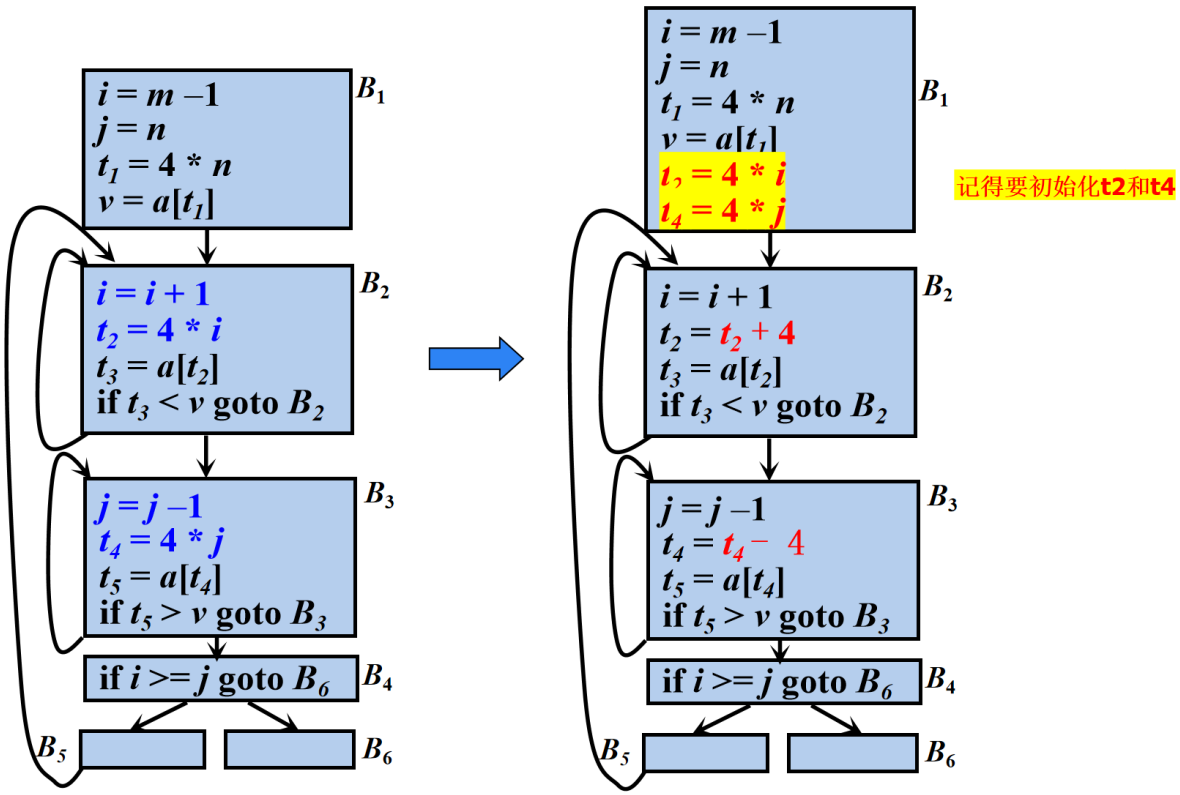
④ 强度削弱

- 用较快的操作代替较慢的操作，如用加代替乘
- 例
  - $2*x$ 或 $2.0*x$ $\Rightarrow$ $x+x$
  - $x/2$ $\Rightarrow$ $x*0.5$
  - x^2 $\Rightarrow$ $x*x$
  - $a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0$ $\Rightarrow((\ldots(a_nx+a_{n-1})x+a_{n-2})\ldots)x+a_1)x+a_0$

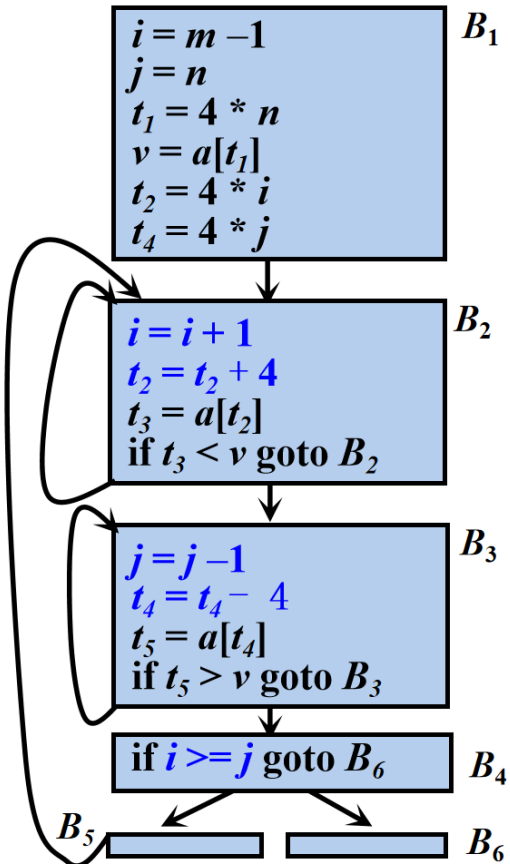
循环中的强度削弱

- 归纳变量
  - 对于一个变量x，如果存在一个正的或负的常数c使得每次x被赋值时它的值总增加c，那么x就称为归纳变量(*Induction Variable*)

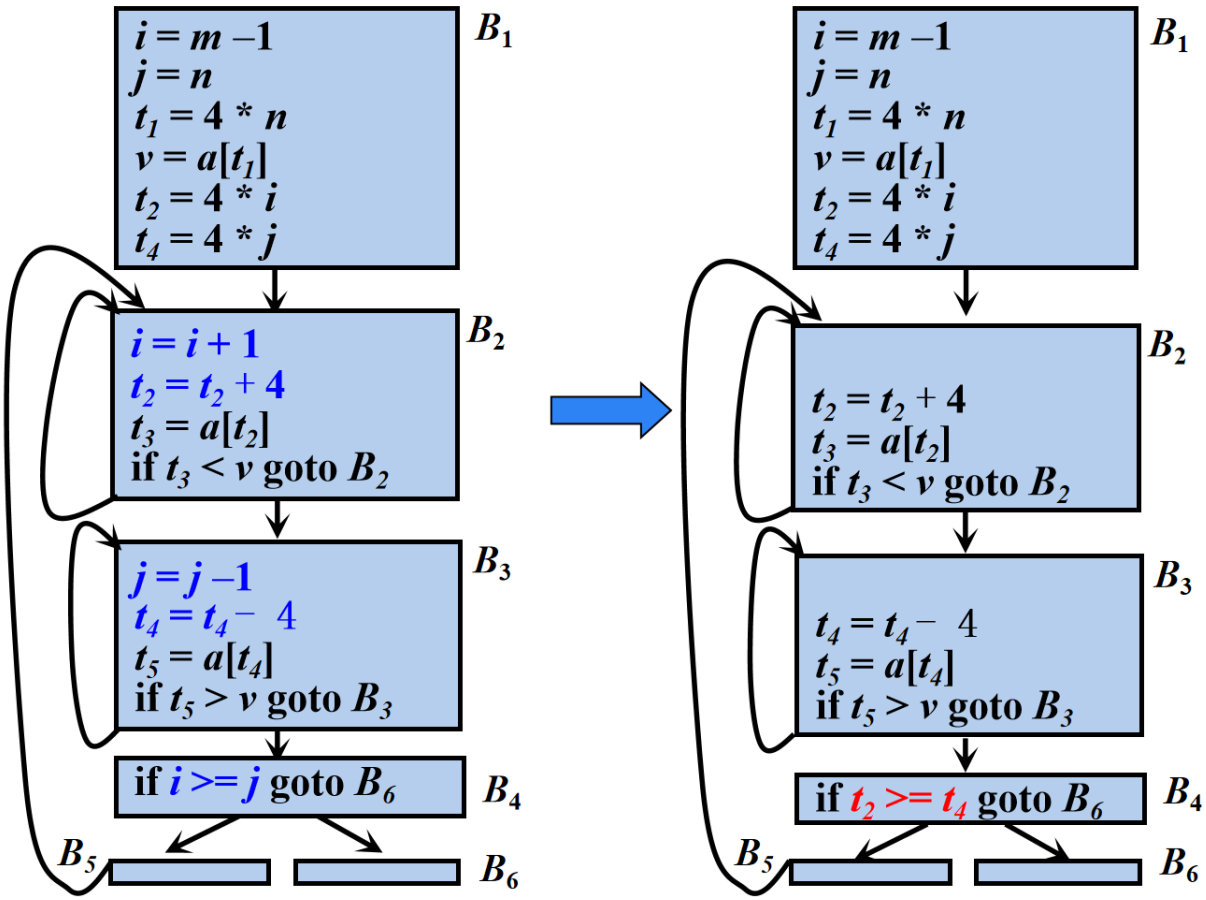
- 归纳变量可以通过在每次循环迭代中进行一次简单的增量运算(加法或减法)来计算



⑤ 删除归纳变量



在沿着循环运行时，如果有一组归纳变量的值的变化保持步调一致，常常可以将这组变量删除为只剩一个



8.3 基本块的优化

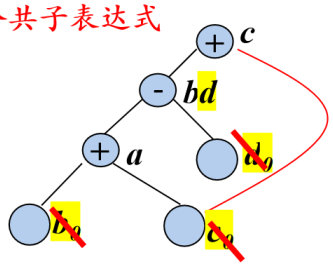
➢ 很多重要的局部优化技术首先把一个基本块转换成为一个无环有向图(*directed acyclic graph*, *DAG*)

基本块的 DAG 表示

- 例

$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

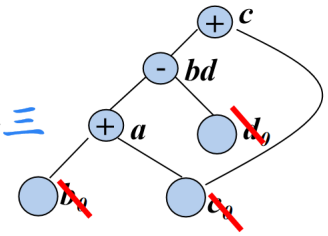
对于形如$x=y+z$的三地址指令，如果已经有一个结点表示$y+z$，就不往DAG中增加新的结点，而是给已经存在的结点附加定值变量x



- 基本块中的每个语句s都对应一个内部结点N
  - 结点N的标号是s中的运算符；同时还有一个定值变量表被关联到N，表示s是在此基本块内最晚对表中变量进行定值的语句
  - N的子结点是基本块中在s之前、最后一个对s所使用的运算分量进行定值的语句对应的结点。如果s的某个运算分量在基本块内没有在s之前被定值，则这个运算分量对应的子结点就是代表该运算分量初始值的叶结点(为区别起见，叶节点的定值变量表中的变量加上下脚标0)
  - 在为语句$x=y+z$构造结点N的时候，如果x已经在某结点M的定值变量表中，则从M的定值变量表中删除变量x

- **DAG图能自动检测公共子表达式**

从 *DAG* 到基本块的重组



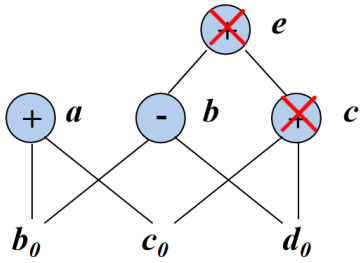
- 对每个具有若干定值变量的节点，构造一个三地址语句来计算其中某个变量的值
  - 倾向于把计算得到的结果赋给一个在基本块出口处活跃的变量(如果没有全局活跃变量的信息作为依据，就要假设所有变量都在基本块出口处活跃，但是不包含编译器为处理表达式而生成的临时变量)
  - 如果结点有多个附加的活跃变量，就必须引入复制语句，以便给每一个变量都赋予正确的值

基于基本块的 DAG 删除无用代码

- 从一个*DAG*上删除所有没有附加活跃变量（活跃变量是指其值可能会在以后被使用的变量）的根结点(即没有父结点的结点)。重复应用这样的处理过程就可以从*DAG*中消除所有对应于无用代码的结点

- 例

$a = b + c$
$b = b - d$
~~$c = c + d$~~
~~$e = b + c$~~

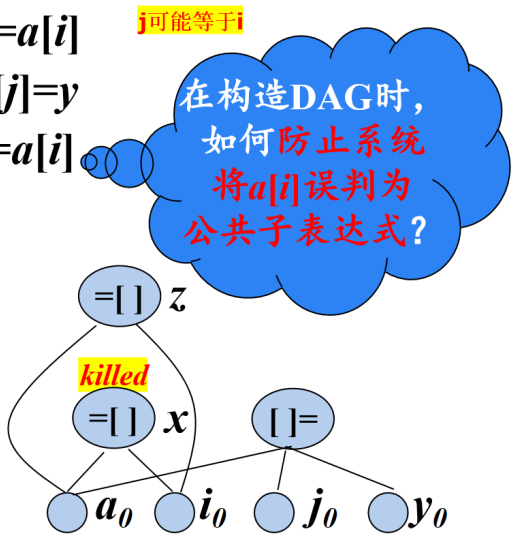


假设*a*和*b*是活跃变量，但*c*和*e*不是

数组元素赋值指令的表示

- 例

$x=a[i]$
$a[j]=y$
$z=a[i]$

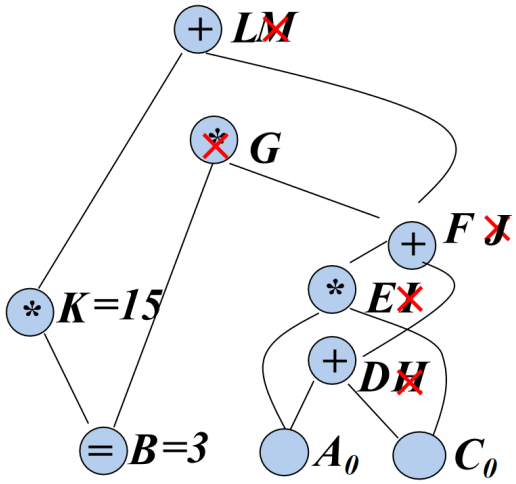


- 对于形如*a*[*j*] = *y*的三地址指令，创建一个运算符为“[]=”的结点，这个结点有3个子结点，分别表示*a*、*j*和*y*
- 该结点没有定值变量表
- 该结点的创建将杀死所有已经建立的、其值依赖于*a*的结点
- 一个被杀死的结点不能再获得任何定值变量，也就是说，它不可能成为一个公共子表达式

例

➢ 给定一个基本块

① $B = 3$
② $D = A + C$
③ $E = A * C$
④ $F = E + D$
⑤ $G = B * F$
⑥ $H = A + C$
⑦ $I = A * C$
⑧ $J = H + I$
⑨ $K = B * 5$
⑩ $L = K + J$
⑪ $M = L$



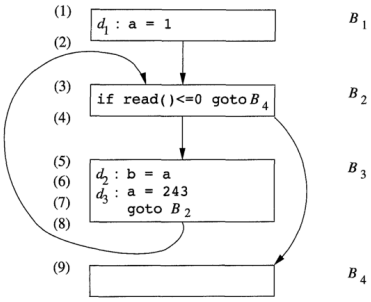
$D = A + C$
$E = A * C$
$F = E + D$
$L = 15 + F$

能计算出常量的直接用常量代替

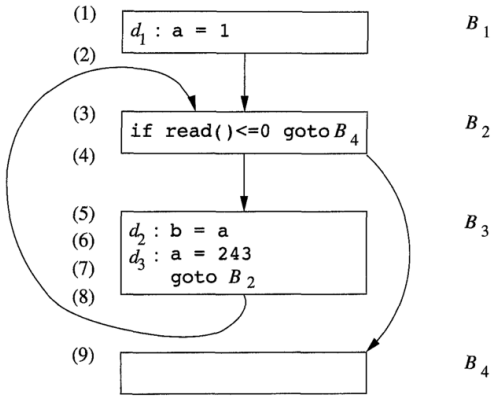
假设：仅变量L在基本块出口之后活跃

8.4 数据流分析

- **数据流分析**
  - 一组用来获取有关数据如何沿着程序执行路径流动的相关信息的技术
- 在每一种数据流分析应用中，都会把每个程序点和一个数据流值关联起来
  - 程序点：流图基本块中的位置，包括
    - 第一个语句之前
    - 两个相邻语句之间
    - 最后一个语句之后



  - 如果有一个从基本块B_1到基本块B_2的边，那么B_2的第一个语句之前的程序点可能紧跟在B_1的最后一个语句的程序点之后



假设所关心的数据流值为：在每个程序点，变量a可能有哪些值

- 程序点(6)：$\{1, 243\}$
- 程序点(7)：$\{243\}$

数据流分析的主要应用

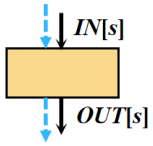
- 到达-定值分析 (Reaching-Definition Analysis)
- 活跃变量分析 (Live-Variable Analysis)
- 可用表达式分析 (Available-Expression Analysis)

数据流分析模式

- 语句的数据流模式
  - *IN*[*s*]: 语句*s*之前的数据流值

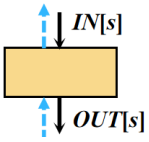
    OUT[*s*]: 语句*s*之后的数据流值
  - f_s: 语句*s*的传递函数(transfer function)
    - 一个赋值语句*s*之前和之后的数据流值的关系
    - 传递函数的两种风格
      - 信息沿执行路径前向传播(前向数据流问题)

        $OUT[s] = f_s(IN[s])$

      - 信息沿执行路径逆向传播(逆向数据流问题)

        $IN[s] = f_s(OUT[s])$

  - 基本块中相邻两个语句之间的数据流值的关系
    - 设基本块*B*由语句$s_1, s_2, \ldots, s_n$顺序组成，则

      $IN[s_{i+1}] = OUT[s_i] \quad i=1, 2, \ldots, n-1$

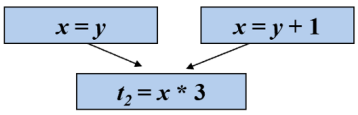
基本块上的数据流模式

- $IN[B]$：紧靠基本块B之前的数据流值
  $OUT[B]$：紧随基本块B之后的数据流值
- 设基本块B由语句$s_1,s_2,\ldots,s_n$顺序组成，则
  - $IN[B] = IN[s_1]$
  - $OUT[B] = OUT[s_n]$
- f_B：基本块B的传递函数
  - 前向数据流问题：$OUT[B] = f_B(IN[B])$
    $f_B = f_{sn} \cdot \ldots \cdot f_{s2} \cdot f_{s1}$
  - 逆向数据流问题：$IN[B] = f_B(OUT[B])$
    $f_B = f_{s1} \cdot f_{s2} \cdot \ldots \cdot f_{sn}$

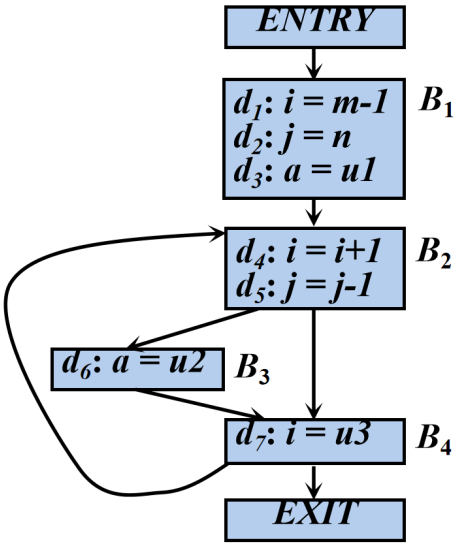
$$
\begin{aligned}
IN[B] &= IN[s_1] \\
&= f_{s1}(OUT[s_1]) \\
&= f_{s1}(IN[s_2]) \\
&= f_{s1} \cdot f_{s2}(OUT[s_2]) \\
&= f_{s1} \cdot f_{s2}(IN[s_3]) \\
&\ldots \\
&= f_{s1} \cdot f_{s2} \cdot \ldots \cdot f_{s(n-1)}(IN[s_n]) \\
&= f_{s1} \cdot f_{s2} \cdot \ldots \cdot f_{s(n-1)} \cdot f_{sn}(OUT[s_n]) \\
&= f_{s1} \cdot f_{s2} \cdot \ldots \cdot f_{s(n-1)} \cdot f_{sn}(OUT[B])
\end{aligned}
$$

8.4.1 到达定值分析

- 定值 (*Definition*)
  - 变量x的定值是(可能)将一个值赋给x的语句
- 到达定值(*Reaching Definition*)　有一条就行，不是一定所有都到达
  - 如果存在一条从紧跟在x的定值d后面的点到达某一程序点p的路径，而且在此路径上d没有被“杀死”(如果在此路径上有对变量x的其它定值d'，则称定值d被定值d'“杀死”了)，则称定值d到达程序点p
  - 直观地讲，如果某个变量x的一个定值d到达点p，在点p处使用的x的值可能就是由d最后赋予的



例：可以到达各基本块的入口处的定值



假设每个控制流图都有两个空基本块，分别是表示流图的开始点的*ENTRY*结点和结束点的*EXIT*结点（所有离开该图的控制流都流向它）

IN[B]	B_2	B_3	B_4
d_1	√	×	×
d_2	√	×	×
d_3	√	√	√
d_4	×	√	√
d_5	√	√	√
d_6	√	√	√
d_7	√	×	×

- 注：定值是语句

到达定值分析的主要用途

- 循环不变计算的检测
  - 如果循环中含有赋值$x=y+z$，而y和z所有可能的定值都在循环外面(包括y或z是常数的特殊情况)，那么$y+z$就是循环不变计算
- 常量传播
  - 如果对变量x的某次使用只有一个定值可以到达，并且该定值把一个常量赋给x，那么可以简单地把x替换为该常量
- 判定变量x在p点上是否未经定值就被引用

“生成”与“杀死”定值

这里，“+” 代表一个一般性的二元运算符

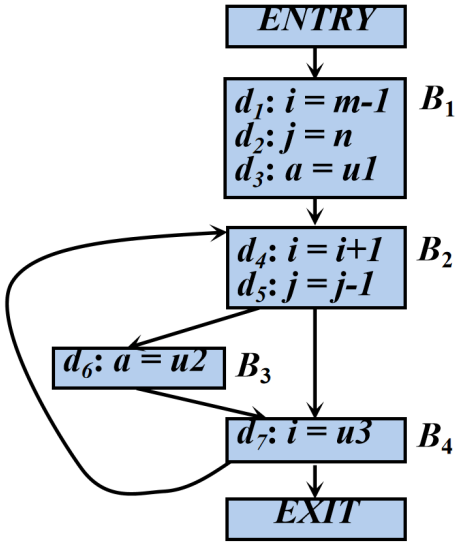
- 定值d: $u = v + w$
  - 该语句“生成”了一个对变量u的定值d，并“杀死”了程序中其它对u的定值

到达定值的传递函数

- f_d： 定值d: $u = v + w$的传递函数
  - $f_d(x) = gen_d \cup (x\text{-}kill_d)$ —— 生成-杀死形式
    - gen_d： 由语句d生成的定值的集合　　$gen_d = \{d\}$
    - $kill_d$： 由语句d杀死的定值的集合（程序中所有其它对u的定值）

- f_B： 基本块B的传递函数
  - $f_B(x) = gen_B \cup (x\text{-}kill_B)$
    - $kill_B = kill_1 \cup kill_2 \cup \ldots \cup kill_n$
      - 被基本块B中各个语句杀死的定值的集合
    - $gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \ldots \cup (gen_1 - kill_2 - kill_3 - \ldots - kill_n)$
      - 基本块中没有被块中各语句“杀死”的定值的集合

例：各基本块B的gen_B和$kill_B$



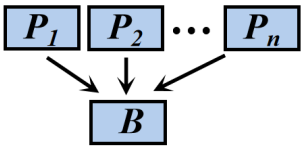
- $gen_{B1}=\{d_1, d_2, d_3\}$
- $kill_{B1}=\{d_4, d_5, d_6, d_7\}$
- $gen_{B2}=\{d_4, d_5\}$
- $kill_{B2}=\{d_1, d_2, d_7\}$
- $gen_{B3}=\{d_6\}$
- $kill_{B3}=\{d_3\}$
- $gen_{B4}=\{d_7\}$
- $kill_{B4}=\{d_1, d_4\}$

到达定值的数据流方程

- $IN[B]$：到达流图中基本块B的入口处的定值的集合

  $OUT[B]$：到达流图中基本块B的出口处的定值的集合
- 方程
  - $OUT[ENRTY]=\Phi$
  - $OUT[B]=f_B(IN[B])$ $(B\neq ENTRY)$
    - $f_B(x)=gen_B \cup (x-kill_B)$

  $OUT[B]=gen_B \cup (IN[B]-kill_B)$
  - $IN[B]=\cup_{P是B的一个前驱} OUT[P]$ $(B\neq ENTRY)$



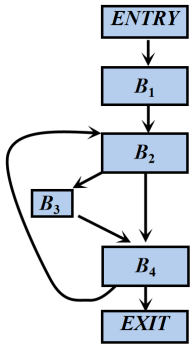
gen_B和$kill_B$的值可以直接从流图计算出来，因此在方程中作为已知量

计算到达定值的迭代算法

- 输入：
  - 流图G，其中每个基本块B的gen_B和$kill_B$都已计算出来
- 输出：
  - $IN[B]$和$OUT[B]$
- 方法：

$OUT[ENTRY] = \Phi$;
for (除$ENTRY$之外的每个基本块B) $OUT[B] = \Phi$;
while (某个OUT值发生了改变)
  for (除$ENTRY$之外的每个基本块B) {
    $IN[B] = \cup_{P是B的一个前驱} OUT[P]$;
    $OUT[B] = gen_B \cup (IN[B] - kill_B)$
  }

例



$gen_{B1}=\{d_1, d_2, d_3\}$
$kill_{B1}=\{d_4, d_5, d_6, d_7\}$
$gen_{B2}=\{d_4, d_5\}$
$kill_{B2}=\{d_1, d_2, d_7\}$
$gen_{B3}=\{d_6\}$
$kill_{B3}=\{d_3\}$
$gen_{B4}=\{d_7\}$
$kill_{B4}=\{d_1, d_4\}$

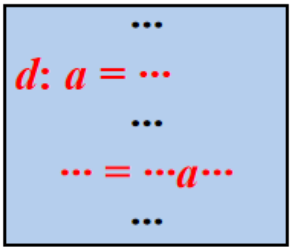
$OUT[ENTRY] = \Phi$;
for (除$ENTRY$之外的每个基本块B) $OUT[B] = \Phi$;
while (某个OUT值发生了改变)
  for (除$ENTRY$之外的每个基本块B) {
    $IN[B] = \cup_{P是B的一个前驱} OUT[P]$;
    $OUT[B] = gen_B \cup (IN[B] - kill_B)$
  }

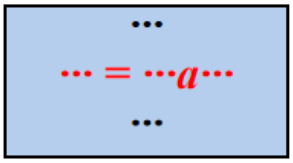
$IN[B]$	B_2	B_3	B_4
d_1	√	×	×
d_2	√	×	×
d_3	√	√	√
d_4	×	√	√
d_5	√	√	√
d_6	√	√	√
d_7	√	×	×

B	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$	$IN[B]^3$	$OUT[B]^3$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111	001 1110	001 0111
$EXIT$	000 0000	001 0111	001 0111	001 0111	001 0111	001 0111	001 0111

引用-定值链

- 引用-定值链(简称ud链)是一个列表，对于变量的每一次引用，到达该引用的所有定值都在该列表中
  - 如果块B中变量a的引用之前有a的定值，那么只有a的最后一次定值会在该引用的ud链中
  - 如果块B中变量a的引用之前没有a的定值，那么a的这次引用的ud链就是IN[B]中a的定值的集合

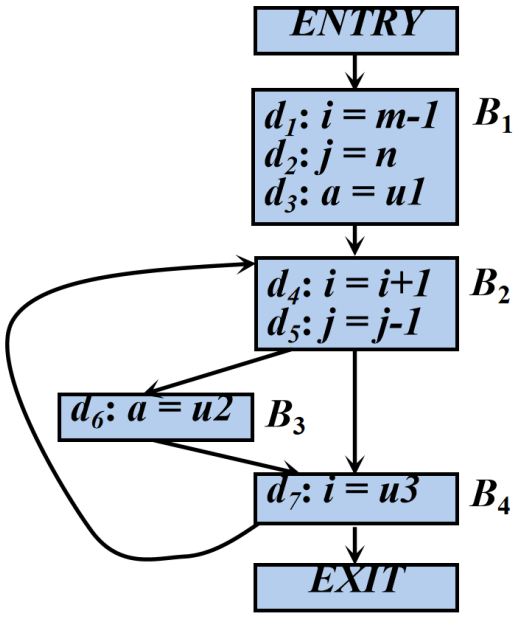




8.4.2 活跃变量分析

- 活跃变量
  - 对于变量x和程序点p，如果在流图中沿着从p开始的某条路径会引用变量x在p点的值，则称变量x在点p是活跃(live)的，否则称变量x在点p不活跃(dead)

例：各基本块的出口处的活跃变量



a一直没有被引用

OUT[B]	B_1	B_2	B_3	B_4
a	×	×	×	×
i	√	×	×	√
j	√	√	√	√
m	×	×	×	×
n	×	×	×	×
u1	×	×	×	×
u2	√	√	√	√
u3	√	√	√	√

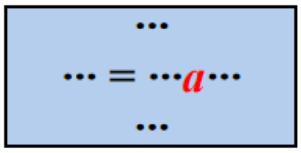
- `i` 离开基本块B2后，无论走哪条路径都会经过B4，在B4被重新定值，且每个路径都没有引用 i ，所以 i 在B2和B3出口不活跃。
- 但从B4走可以到达B2，`i` 被引用了，所以在B4出口处活跃
- `j` 从B2离开后还可以回到B2，B2中有对`j`的引用，且路径中没有被重新定值，所以`j` 在各个块的出口处都是活跃的
- `m、n、u1` 在后续都没有被引用
- 控制从各个基本块离开后都可以再进入B3、B4，且`u2、u3` 都没有被重新定值，所以在各个控制块的出口处都活跃

活跃变量信息的主要用途

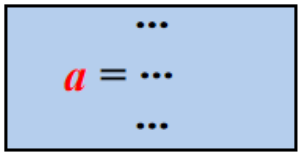
- 删除无用赋值
  - 无用赋值：如果x在点p的定值在基本块内所有后继点都不被引用，且x在基本块出口之后又是不活跃的，那么x在点p的定值就是无用的
- 为基本块分配寄存器
  - 如果所有寄存器都被占用，并且还需要申请一个寄存器，则应该考虑使用已经存放了死亡值的寄存器，因为这个值不需要保存到内存
  - 如果一个值在基本块结尾处是死的就不必在结尾处保存这个值

活跃变量的传递函数

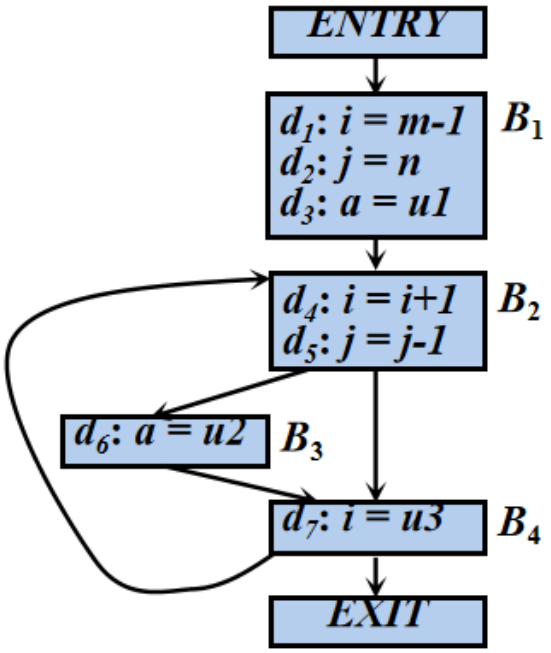
- 逆向数据流问题
  - $IN[B] = f_B(OUT[B])$



- $f_B(x) = use_B \cup (x - def_B)$
  - use_B：在基本块B中引用，但是引用前在B中没有被定值的变量集合
  - def_B：在基本块B中定值，但是定值前在B中没有被引用的变量的集合



例：各基本块B的use_B和def_B



- use_{B1} ={ m, n, u1 }
- def_{B1} ={ i, j, a }
- use_{B2} ={ i, j }
- def_{B2}= Φ
- use_{B3} ={ u2 }
- def_{B3} ={ a }
- use_{B4} ={ u3 }
- def_{B4} ={ i }

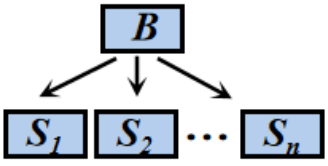
活跃变量数据流方程

- $IN[B]$：在基本块B的入口处的活跃变量集合
  $OUT[B]$：在基本块B的出口处的活跃变量集合
- 方程
  - $IN[EXIT] = \Phi$
  - $IN[B] = f_B(OUT[B])$ （$B \neq EXIT$）
    - $f_B(x) = use_B \cup (x - def_B)$

  $IN[B] = use_B \cup (OUT[B] - def_B)$

  - $OUT[B] = \cup_{S是B的一个后继} IN[S]$ （$B \neq EXIT$）

use_B和def_B的值可以直接从流图计算出来，因此在方程中作为已知量

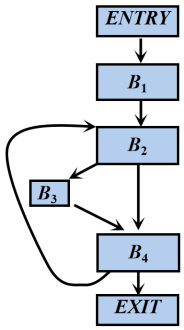


计算活跃变量的迭代算法

- 输入：流图G，其中每个基本块B的use_B和def_B都已计算出来
- 输出：$IN[B]$和$OUT[B]$
- 方法：

```
IN[EXIT] = Φ;
for (除EXIT之外的每个基本块B)  IN[B] = Φ;
while (某个IN值发生了改变)
   for (除EXIT之外的每个基本块B) {
         OUT[B] = ∪ S是B的一个后继 IN[S];
         IN[B] = use_B ∪ (OUT[B] - def_B );
   }
```

例



use_{B1} ={ m, n, u1 }
def_{B1} ={ i, j, a }
use_{B2} ={ i, j }
def_{B2}= Φ
use_{B3} ={ u2 }
def_{B3} ={ a }
use_{B4} ={ u3 }
def_{B4} ={ i }

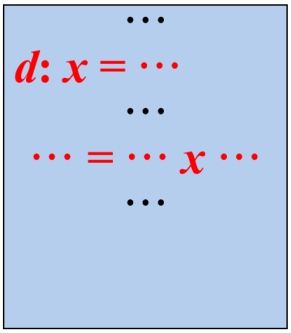
```
IN[EXIT] = Φ;
for (除EXIT之外的每个基本块B)  IN[B] = Φ;
while (某个IN值发生了改变)
   for (除EXIT之外的每个基本块B) {
         OUT[B] = ∪ S是B的一个后继 IN[S];
         IN[B] = use_B ∪ (OUT[B] - def_B );
   }
```

	$OUT[B]^1$	$IN[B]^1$	$OUT[B]^2$	$IN[B]^2$	$OUT[B]^3$	$IN[B]^3$
B_4		u3	i,j,u2,u3	j,u2,u3	i,j,u2,u3	j,u2,u3
B_3	u3	u2,u3	j,u2,u3	j,u2,u3	j,u2,u3	j,u2,u3
B_2	u2,u3	i,j,u2,u3	j,u2,u3	i,j,u2,u3	j,u2,u3	i,j,u2,u3
B_1	i,j,u2,u3	m,n,u1,u2,u3	i,j,u2,u3	m,n,u1,u2,u3	i,j,u2,u3	m,n,u1,u2,u3

$OUT[B]$	B_1	B_2	B_3	B_4
a	×	×	×	×
i	√	×	×	√
j	√	√	√	√
m	×	×	×	×
n	×	×	×	×
u_1	×	×	×	×
u_2	√	√	√	√
u_3	√	√	√	√

定值-引用链

- 定值-引用链：设变量x有一个定值d，该定值所有能够到达的引用u的集合称为x在d处的定值-引用链，简称du链
- 如果在求解活跃变量数据流方程中的$OUT[B]$时，将$OUT[B]$表示成从B的末尾处能够到达的引用的集合，那么，可以直接利用这些信息计算基本块B中每个变量x在其定值处的du链
  - 如果B中x的定值d之后有x的第一个定值d'，则d和d'之间x的所有引用构成d的du链
  - 如果B中x的定值d之后没有x的新的定值，则B中d之后x的所有引用以及$OUT[B]$中x的所有引用构成d的du链



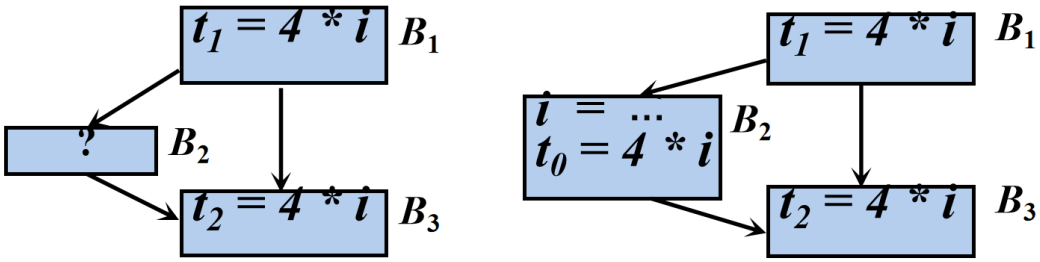
8.4.3 可用表达式分析

- 可用表达式
  - 如果从流图的首节点到达程序点p的每条路径都对表达式x op y进行计算，并且从最后一个这样的计算到点p之间没有再次对x或y定值，那么表达式x op y在点p是可用的(*available*)
- 表达式可用的直观意义
  - 在点p上，x op y已经在之前被计算过，不需要重新计算

可用表达式信息的主要用途

➢消除全局公共子表达式

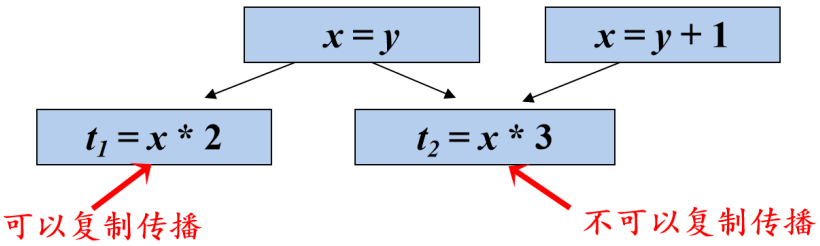
➢例



如果i在B_2中没有被赋予新值，或者在B_2中，对i赋值后又重新计算了$4*i$

➢进行复制传播

➢例



在x的引用点u可以用y代替x的条件：复制语句$x = y$在引用点u处可用

从流图的首节点到达u的每条路径都存在复制语句$x = y$，并且从最后一条复制语句$x = y$到点u之间没有再次对x或y定值

可用表达式的传递函数

- 对于可用表达式数据流模式而言，如果基本块B对x op y进行计算，并且之后没有重新定值x或y，则称B生成表达式x op y；如果基本块B对x或者y进行了(或可能进行)定值，且以后没有重新计算x op y，则称B杀死表达式x op y

- $f_B(x)= e\_gen_B \cup (x- e\_kill_B)$
  - e_gen_B：基本块B所生成的可用表达式的集合
  - e_kill_B：基本块B所杀死的U中的可用表达式的集合
    - U：所有出现在程序中一个或多个语句的右部的表达式的全集

e_genB的计算

- 初始化：$e\_gen_B = \Phi$
- 顺序扫描基本块的每个语句：$z = x$ op y
  - 把x op y加入e_gen_B
  - 从e_gen_B中删除和z相关的表达式

顺序不能颠倒
例：$x = x$ op y

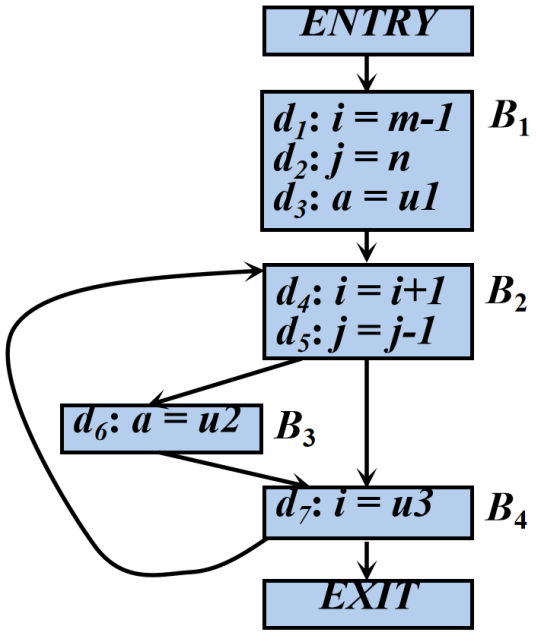
语句	可用表达式
……	Ø
a : = b+c	
……	{ b+c }
b : = a-d	
……	{ a-d }
c : = b+c	
……	{ a-d }
d : = a-d	
……	Ø

e_killB的计算

- 初始化：$e\_kill_B = \Phi$
- 顺序扫描基本块的每个语句：$z = x \text{ op } y$
  - 从e_kill_B 中删除表达式$x \text{ op } y$
  - 把所有和z相关的表达式加入到e_kill_B 中

例：各基本块B的e_gen和e_kill_B

只有表达式，单个的赋值语句不算



- $e\_gen_{B1} = \{ m\text{-}1 \}$
- $e\_kill_{B1} = \{ i\text{+}1, j\text{-}1 \}$
- $e\_gen_{B2} = \Phi$
- $e\_kill_{B2} = \{ i\text{+}1, j\text{-}1 \}$
- $e\_gen_{B3} = \Phi$
- $e\_kill_{B3} = \Phi$
- $e\_gen_{B4} = \Phi$
- $e\_kill_{B4} = \{ i\text{+}1 \}$

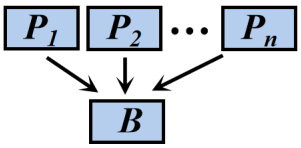
可用表达式的数据流方程

- $IN[B]$：在B的入口处可用的U中的表达式集合

  $OUT[B]$：在B的出口处可用的U中的表达式集合
- 方程
  - $OUT[ENTRY]=\Phi$
  - $OUT[B]=f_B(IN[B])$　$(B\neq ENTRY)$
    - $f_B(x)=e\_gen_B \cup (x-e\_kill_B)$

    $OUT[B]=e\_gen_B \cup (IN[B]-e\_kill_B)$
  - $IN[B]=\cap_{P是B的一个前驱} OUT[P]$　$(B\neq ENTRY)$

e_gen_B和e_kill_B的值可以直接从流图计算出来，因此在方程中作为已知量



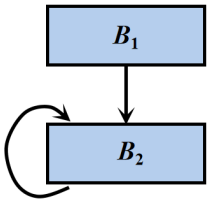
计算可用表达式的迭代算法

- 输入：流图G，其中每个基本块B的e_gen_B和e_kill_B都已计算出来
- 输出：$IN[B]$和$OUT[B]$
- 方法:

```
OUT[ENTRY] = Φ;
for (除ENTRY之外的每个基本块B) OUT[B] = U;
while (某个OUT值发生了改变)
   for (除ENTRY之外的每个基本块B) {
         IN[B]= ∩P是B的一个前驱 OUT[P]
         OUT[B]= e_genB ∪(IN[B]- e_killB) ;
         }
```

为什么将$OUT[B]$集合初始化为U?

- 将OUT集合初始化为Φ局限性太大
  - 例



$$IN[B_2]^1 = OUT[B_1]^1 \cap OUT[B_2]^0$$

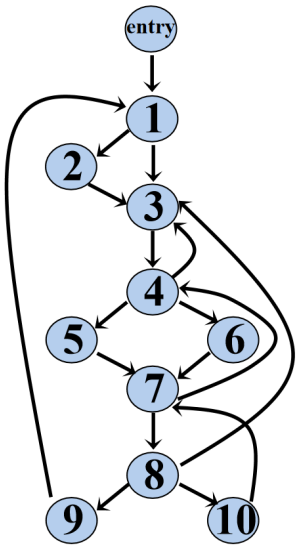
$$= \begin{cases} \Phi & \text{，如果 } OUT[B_2]^0 = \Phi \\ OUT[B_1]^1 & \text{，如果 } OUT[B_2]^0 = U \end{cases}$$

8.5 流图中的循环

- 支配结点 (*Dominators*)
  - 如果从流图的入口结点到结点n的每条路径都经过结点d，则称结点d支配(*dominate*)结点n，记为$d\ dom\ n$

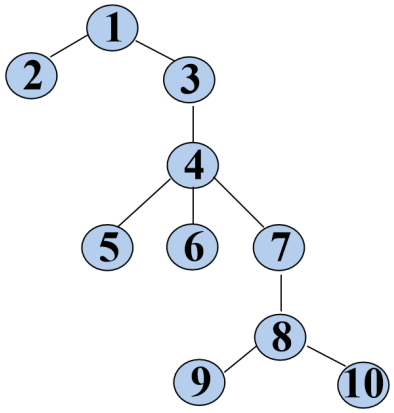
每个结点都支配它自己

例



支配结点	支配对象
1	1～10
2	2
3	3～10
4	4～10
5	5
6	6
7	7～10
8	8～10
9	9
10	10

➢ 支配结点树 (*Dominator Tree*)



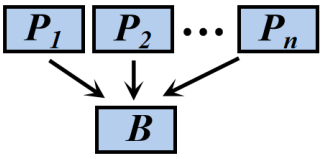
每个结点只支配它和它的后代结点

寻找支配结点

➢ 支配结点的数据流方程

➢ *IN*[*B*]：在基本块*B*入口处的支配结点集合

OUT[*B*]：在基本块*B*出口处的支配结点集合



➢ 方程

➢ $OUT[ENTRY] = \{ ENTRY \}$

➢ $OUT[B] = f_B(IN[B])$ ($B \neq ENTRY$)

➢ $f_B(x) = x \cup \{ B \}$

$OUT[B] = IN[B] \cup \{B\}$

➢ $IN[B] = \cap_{P是B的一个前驱} OUT[P]$ ($B \neq ENTRY$)

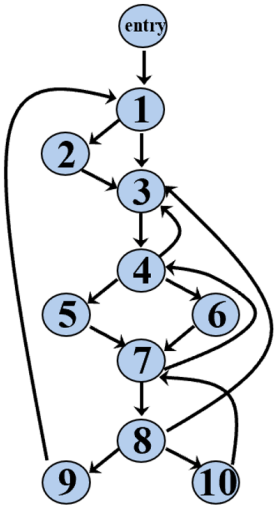
计算支配结点的迭代算法

- 输入：流图G，G的结点集是N，边集是E，入口结点是$ENTRY$
- 输出：对于N中的各个结点n，给出$D(n)$，即支配n的所有结点的集合
- 方法：

```
OUT[ENTRY]={ENTRY}
for(除ENTRY之外的每个基本块B)
   OUT[B]=N
while(某个OUT值发生了改变)
   for(除ENTRY之外的每个基本块B)
      { IN[B]=∩P是B的一个前驱OUT[P]
        OUT[B]=IN[B]∪{B}
      }
```

例

```
OUT[ENTRY]={ENTRY}
for(除ENTRY之外的每个基本块B) OUT[B]=N
while(某个OUT值发生了改变)
    for(除ENTRY之外的每个基本块B)
   { IN[B]=∩P是B的一个前驱OUT[P]
     OUT[B]=IN[B]∪{B}
   }
```

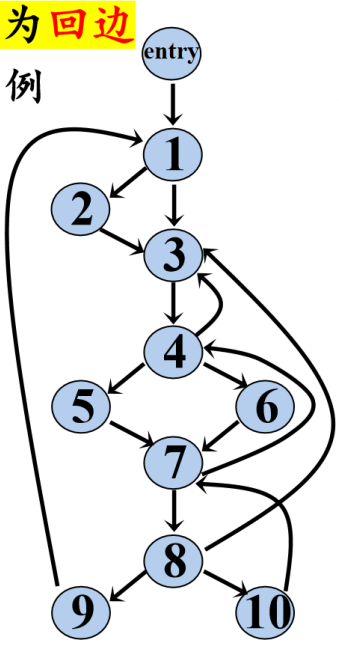


	$OUT^0[B]$	$IN^1[B]$	$OUT^1[B]$
E	$\{E\}$		
①	N	$\{E\}$	$\{E①\}$
②	N	$\{E①\}$	$\{E①②\}$
③	N	$\{E①\}$	$\{E①③\}$
④	N	$\{E①③\}$	$\{E①③④\}$
⑤	N	$\{E①③④\}$	$\{E①③④⑤\}$
⑥	N	$\{E①③④\}$	$\{E①③④⑥\}$
⑦	N	$\{E①③④\}$	$\{E①③④⑦\}$
⑧	N	$\{E①③④⑦\}$	$\{E①③④⑦⑧\}$
⑨	N	$\{E①③④⑦⑧\}$	$\{E①③④⑦⑧⑨\}$
⑩	N	$\{E①③④⑦⑧\}$	$\{E①③④⑦⑧⑩\}$

回边

➢ 如果存在从结点n到d的有向边$n \to d$，且d dom n，那么这条边称为回边

➢ 例



n d

B	$OUT^0[B]$	$IN^1[B]$	$OUT^1[B]$
E	E		
①	N	E	E ①
②	N	E ①	E ① ②
③	N	E ①	E ① ③
④	N	E ① ③	E ① ③ ④
⑤	N	E ① ③ ④	E ① ③ ④ ⑤
⑥	N	E ① ③ ④	E ① ③ ④ ⑥
⑦	N	E ① ③ ④	E ① ③ ④ ⑦
⑧	N	E ① ③ ④ ⑦	E ① ③ ④ ⑦ ⑧
⑨	N	E ① ③ ④ ⑦ ⑧	E ① ③ ④ ⑦ ⑧ ⑨
⑩	N	E ① ③ ④ ⑦ ⑧	E ① ③ ④ ⑦ ⑧ ⑩

回边：

4->3

7->4

8->3

9->1

10->7

自然循环

➢ 从程序分析的角度来看，循环在代码中以什么形式出现并不重要，重要的是它是否具有易于优化的性质

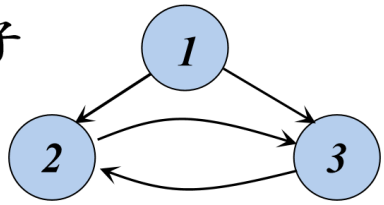
➢ 自然循环是一种适合于优化的循环，它满足以下性质

➢ 有唯一的入口结点，称为首结点(*header*)

➢ 首结点支配循环中的所有结点

➢ 循环中至少有一条返回首结点的路径，否则，控制就不可能从“循环”中直接回到循环头，也就无法构成循环

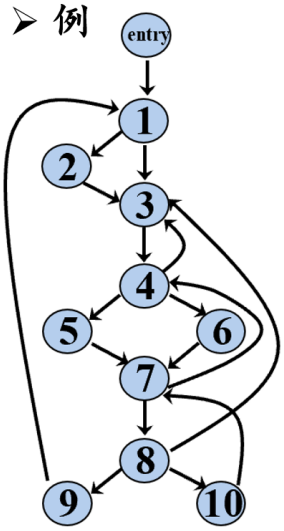
➢ 非自然循环的例子



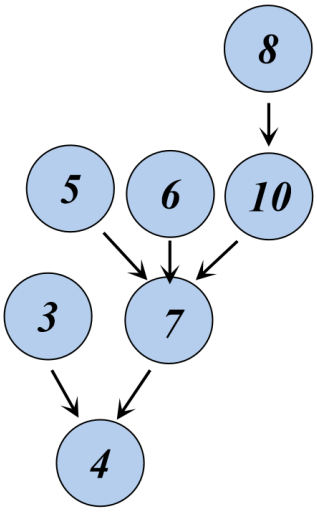
自然循环的识别

➢ 给定一个回边$n \to d$，该回边的自然循环为：d，以及所有可以不经过d而到达n的结点。d为该循环的首结点。

➢ 例



回边	自然循环
4→3	③④⑤⑥⑦⑧⑩
7→4	④⑤⑥⑦⑧⑩
8→3	③④⑤⑥⑦⑧⑩
9→1	① ~ ⑩
10→7	⑦⑧⑩

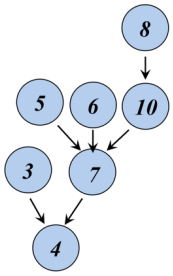


算法：构造一条回边的自然循环

➢ 输入：流图 G和回边$n \to d$

➢ 输出：由回边$n \to d$的自然循环中的所有结点组成的集合

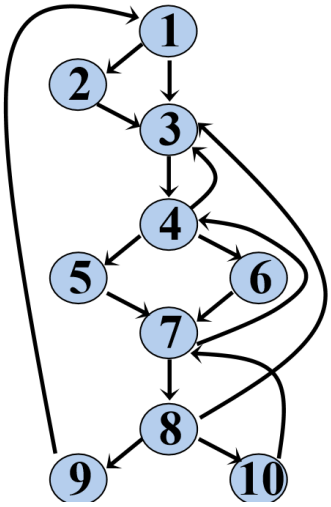
➢ 方法：




```
stack= Φ; loop={ n, d }; push(stack, n);
while stack不空do
{  m = top(stack); pop(stack);
   for m的每个前驱p
   {  if p不在loop中 then
      {  loop = loop ∪{ p}; push(stack,p); }
   }
}
```

结点d在初始时刻已经在$loop$中，不会把它放入栈中，不会去考虑它的前驱。因此，找出的都是不经过d而能到达n的结点。

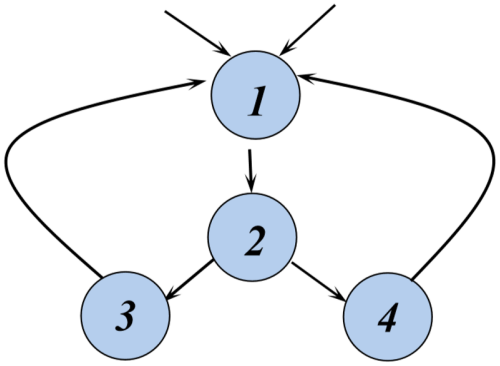
- **自然循环的一个重要性质**
  - 如果两个自然循环的首结点不相同，则这两个循环要么互不相交，要么一个完全包含(嵌入)在另外一个里面
    - 例



回边	自然循环
4->3	③④⑤⑥⑦⑧⑩
7->4	④⑤⑥⑦⑧⑩
8->3	③④⑤⑥⑦⑧⑩
9->1	① ~ ⑩
10->7	⑦⑧⑩

最内循环 (*Innermost Loops*):
不包含其它循环的循环

  - 如果两个循环具有相同的首结点，那么很难说哪个是最内循环。此时把两个循环合并



数据流分析的主要应用

- **到达定值** ∃
  - 定义：如果存在一条从紧跟在x的定值d后面的点到达某一程序点p的路径，而且在此路径上d没有被“杀死”(如果在此路径上有对变量x的其它定值d'，则称定值d被定值d'“杀死”了)，则称定值d到达程序点p
  - 分析任务：程序点p处使用的x的值可能是在哪里定值的
  - 分析结果的表示：*ud*链(引用-定值链)
  - 主要用途
    - 循环不变计算的检测
    - 检测“变量未经定值就被引用”
    - 常量传播
- **活跃变量** ∃
  - 对于变量x和程序点p，如果在流图中沿着从p开始的某条路径会引用变量x在p点的值，则称变量x在点p是活跃(*live*)的，否则称变量x在点p不活跃(*dead*)
  - 分析任务：x在程序点p处的定值在哪里会被使用
  - 分析结果的表示：*du*链(定值-引用链)
  - 主要用途
    - 删除无用赋值
    - 寄存器分配

- **可用表达式** ∀
  - 如果从流图的首节点到达程序点 p 的每条路径都对表达式 x op y 进行计算，并且从最后一个这样的计算到点 p 之间没有再次对 x 或 y 定值，那么表达式 x op y 在点 p 是可用的(*available*)
  - 主要用途
    - 消除全局公共子表达式
    - 复制传播
- **支配结点** ∀
  - 如果从流图的入口结点到结点 n 的每条路径都经过结点 d，则称结点 d 支配(*dominate*)结点 n，记为 d *dom* n

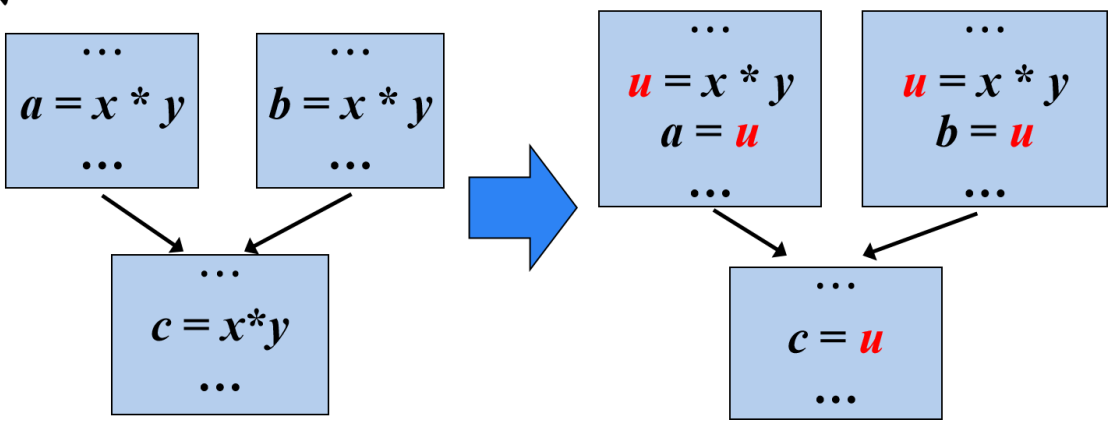
- 到达定值 ∃ 正向
- 活跃变量 ∃ 逆向
- 可用表达式 ∀ 正向
- 支配结点 ∀ 正向

$OUT[ENTRY] = \Phi$ $OUT[B] = \Phi$	$IN[B] = \cup_{P是B的一个前驱} OUT[P]$ $OUT[B] = gen_B \cup (IN[B] - kill_B)$
$IN[EXIT] = \Phi$ $IN[B] = \Phi$	$OUT[B] = \cup_{S是B的一个后继} IN[S]$ $IN[B] = use_B \cup (OUT[B] - def_B)$
$OUT[ENTRY] = \Phi$; $OUT[B] = U$;	$IN[B] = \cap_{P是B的一个前驱} OUT[P]$ $OUT[B] = e\_gen_B \cup (IN[B] - e\_kill_B)$
$OUT[ENTRY] = \{ENTRY\}$ $OUT[B] = N$	$IN[B] = \cap_{P是B的一个前驱} OUT[P]$ $OUT[B] = IN[B] \cup \{B\}$

8.6 全局优化

① 删除全局公共子表达式

- 可用表达式的数据流问题可以帮助确定位于流图中p点的表达式是否为全局公共子表达式
  - 例



全局公共子表达式删除算法

- 输入：带有可用表达式信息的流图
- 输出：修正后的流图
- 方法：
  - 对于语句s：$z=x\ \mathrm{op}\ y$，如果$x\ \mathrm{op}\ y$在s之前可用，那么执行如下步骤：

    ① 从s开始逆向搜索，但不穿过任何计算了$x\ \mathrm{op}\ y$的块，找到所有离s最近的计算了$x\ \mathrm{op}\ y$的语句

    ② 建立新的临时变量u

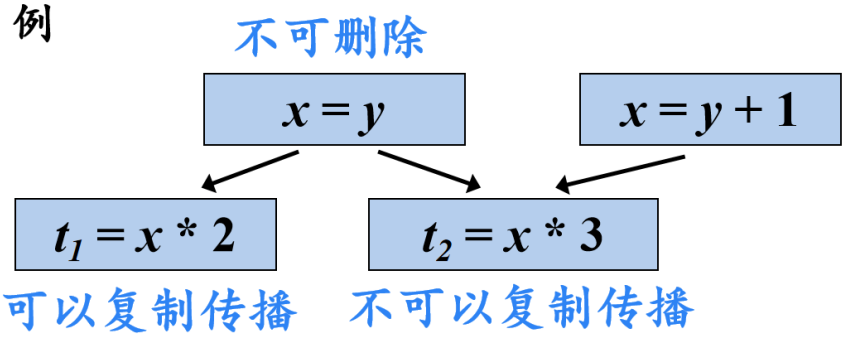
    ③ 把步骤①中找到的语句$w=x\ \mathrm{op}\ y$用下列语句代替：

    $$u=x\ \mathrm{op}\ y$$
    $$w=u$$

    ④ 用$z=u$替代s

② 删除复制语句

- 对于复制语句s: x=y，如果在x的所有引用点都可以用对y的引用代替对x的引用(复制传播)，那么可以删除复制语句 x=y
  - 例



- 在x的引用点u用y代替x (复制传播)的条件
  - 复制语句s: x=y在u点“可用”

删除复制语句的算法

- 输入：流图G、du链、各基本块B入口处的可用复制语句集合
- 输出：修改后的流图
- 方法：
  - 对于每个复制语句x=y，执行下列步骤
    ① 根据du链找出该定值所能够到达的那些对x的引用
    ② 确定是否对于每个这样的引用，x=y都在 IN[B] 中(B是包含这个引用的基本块)，并且B中该引用的前面没有x或者y的定值
    ③ 如果x=y满足第②步的条件，删除x=y，且把步骤①中找到的对x的引用用y代替

③ 代码移动

- 循环不变计算的检测+代码外提

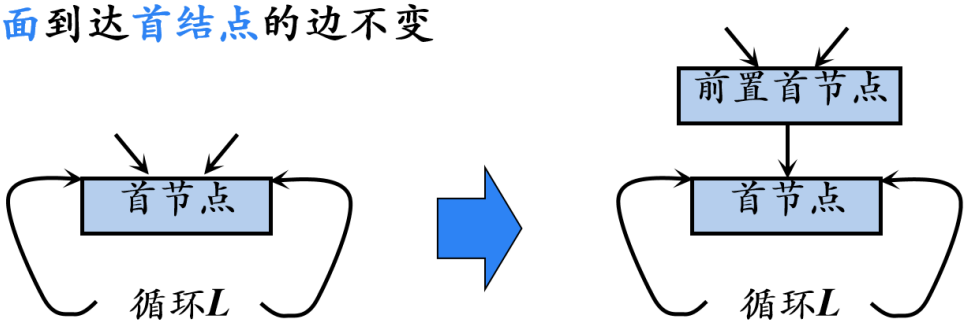
循环不变计算检测算法

- 输入：循环L，每个三地址指令的ud链
- 输出：L的循环不变计算语句
- 方法

1. 将下面这样的语句标记为“不变”：语句的各运算分量或者是常数，或者其所有定值点都在循环L外部
2. 重复执行步骤(3)，直到某次没有新的语句可标记为“不变”为止
3. 将下面这样的语句标记为“不变”：先前没有被标记过，且各运算分量或者是常数，或者其所有定值点都在循环L外部，或者只有一个到达定值，该定值是循环中已经被标记为“不变”的语句

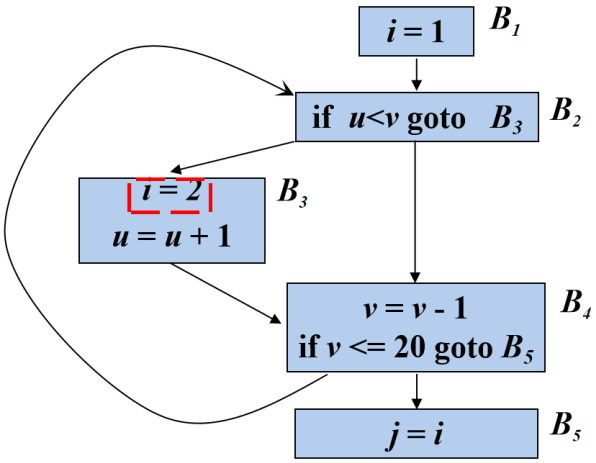
代码外提

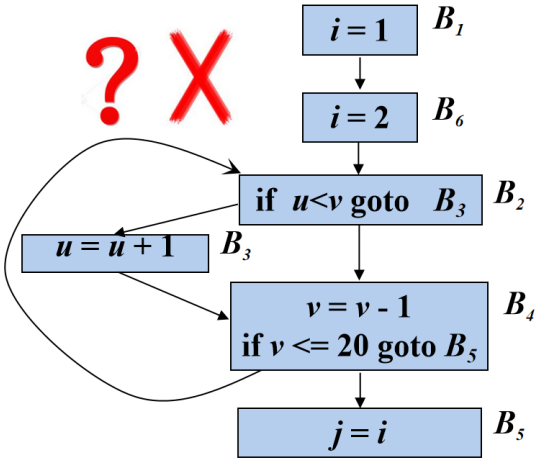
- **前置首结点** (*preheader*)
  - 循环不变计算将被移至首结点之前，为此创建一个称为前置首结点的新块。前置首结点的唯一后继是首结点，并且原来从循环L外到达L首结点的边都改成进入前置首结点。从循环L里面到达首结点的边不变



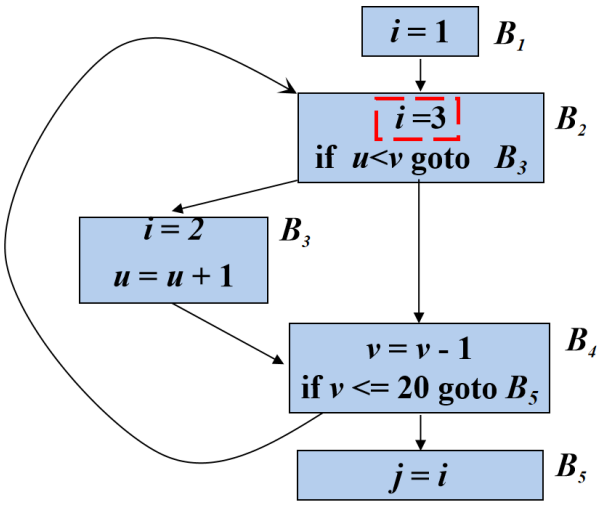
循环不变计算语句 s : x = y + z 移动的条件

(1) s所在的基本块是循环所有出口结点(有后继结点在循环外的结点)的支配结点



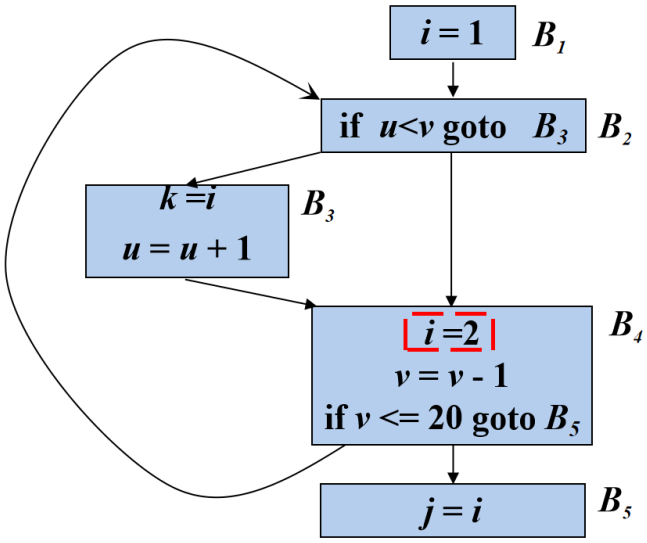


(2) 循环中没有其它语句对x赋值



- 外提前
  - j的值是否等于2取决于循环最后一次迭代时，是否执行了B_3
- 外提后
  - 只要B_3执行过一次，j的值就等于2

(3) 循环中对x的引用仅由s到达

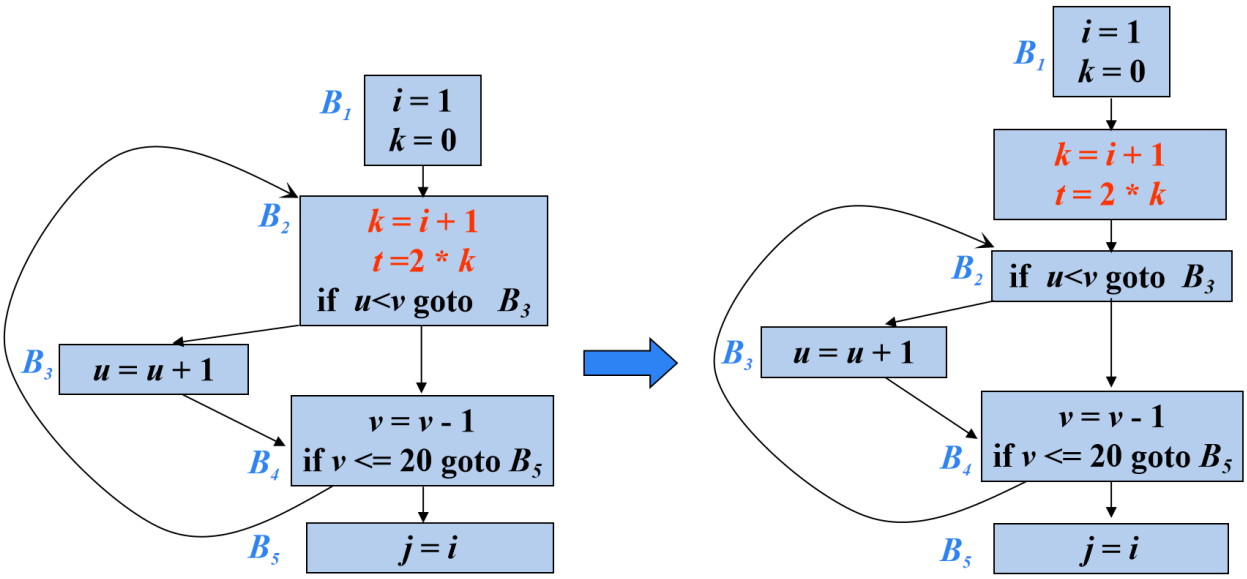


- 外提前
  - k的值有可能等于1，也可能等于2
- 外提后
  - k的值只能等于2

代码移动算法

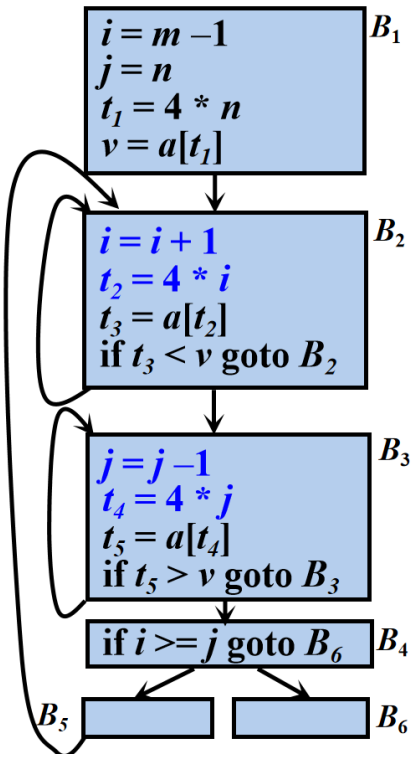
- 输入：循环L、ud链、支配结点信息
- 输出：修改后的循环
- 方法：

1. 寻找循环不变计算
2. 对于步骤(1)中找到的每个循环不变计算，检查是否满足上面的三个条件
3. 按照循环不变计算找出的次序，把所找到的满足上述条件的循环不变计算外提到前置首结点中。如果循环不变计算有分量在循环中定值，只有将定值点外提后，该循环不变计算才可以外提



④ 作用于归纳变量的强度削弱

- 对于一个变量x，如果存在一个正的或负的常量c，使得每次x被赋值时，它的值总是增加c，则称x为归纳变量
- 如果循环L中的变量i只有形如$i=i+c$的定值(c是常量)，则称i为循环L的基本归纳变量
- 如果$j=c\times i+d$，其中i是基本归纳变量，c和d是常量，则j也是一个归纳变量，称j属于i族
  - 基本归纳变量i属于它自己的族
- 每个归纳变量都关联一个三元组。如果$j=c\times i+d$，其中i是基本归纳变量，c和d是常量，则与j相关联的三元组是(i, c, d)



*i*是基本归纳变量

t_2是*i*族归纳变量，可以表示为(*i*, 4, 0)

*j*是基本归纳变量

t_4是*j*族归纳变量，可以表示为(*j*, 4, 0)

归纳变量检测算法

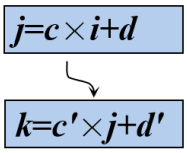
- 输入：带有循环不变计算信息和到达定值信息的循环*L*
- 输出：一组归纳变量
- 方法：

1. 扫描*L*的语句，找出所有基本归纳变量。在此要用到循环不变计算信息。与每个基本归纳变量*i*相关联的三元组是(*i*, 1, 0)

2: 寻找L中只有一次定值的变量k，它具有下面的形式：$k=c'\times j+d'$。其中c'和d'是常量，j是基本的或非基本的归纳变量

- 如果j是基本归纳变量，那么k属于j族。k对应的三元组可以通过其定值语句确定
- 如果j不是基本归纳变量，假设其属于i族，k的三元组可以通过j的三元组和k的定值语句来计算，此时我们还要求：
  - 循环L中对j的唯一定值和对k的定值之间没有对i的定值
  - 循环L外没有j的定值可以到达k

这两个条件是为了保证对k进行赋值的时候，j当时的值一定等于$c*(i$当时的值$)+d$



作用于归纳变量的强度削弱算法

- 输入：带有到达定值信息和已计算出的归纳变量族的循环L
- 输输出：修改后的循环
- 方法：对于每个基本归纳变量i，对其族中的每个归纳变量j：(i, c, d)执行下列步骤

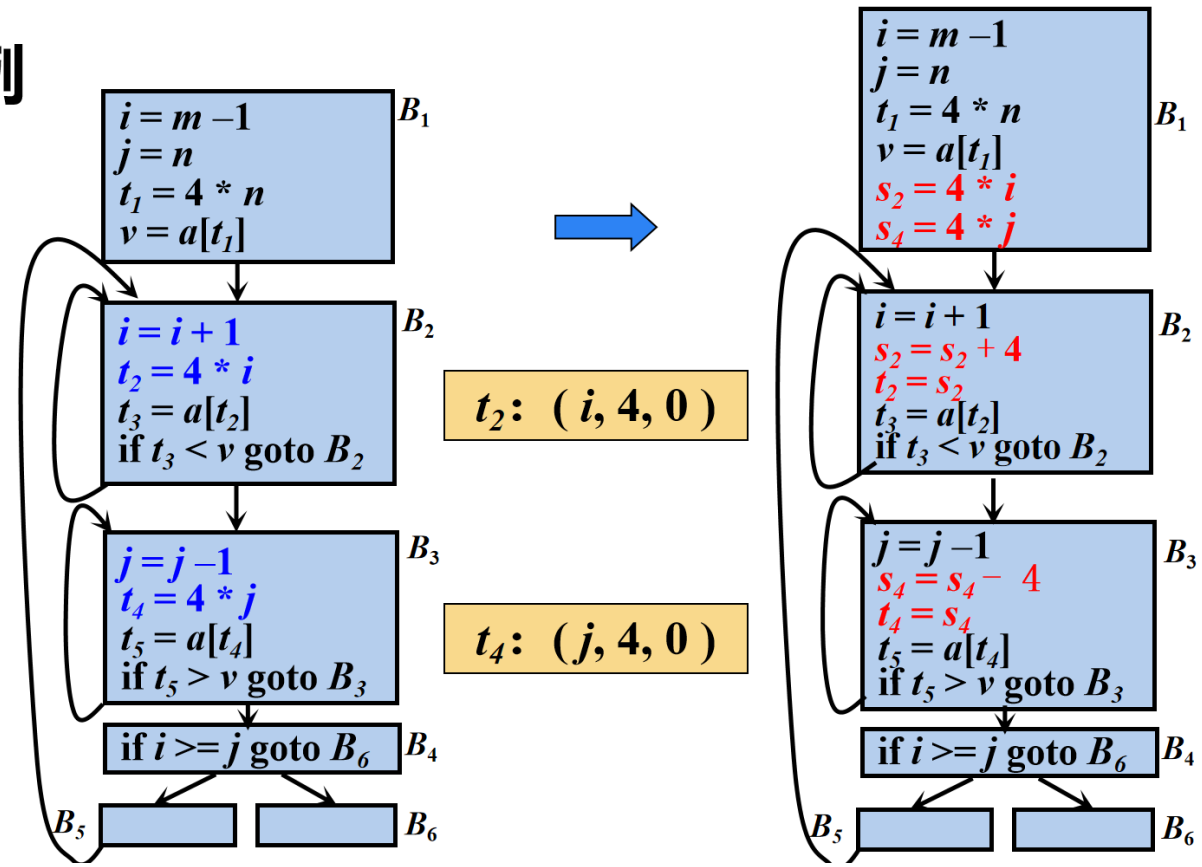
1.建立新的临时变量t。如果变量j_1和j_2具有相同的三元组，则只为它们建立一个新变量

2.在前置节点的末尾，添加语句$t=c*i$和$t=t+d$，使得在循环开始的时候$t=c*i+d=j$

3.在L中紧跟定值$i=i+n$之后，添加$t=t+c*n$。将t放入i族，其三元组为(i, c, d)

4.用$j=t$代替对j的赋值

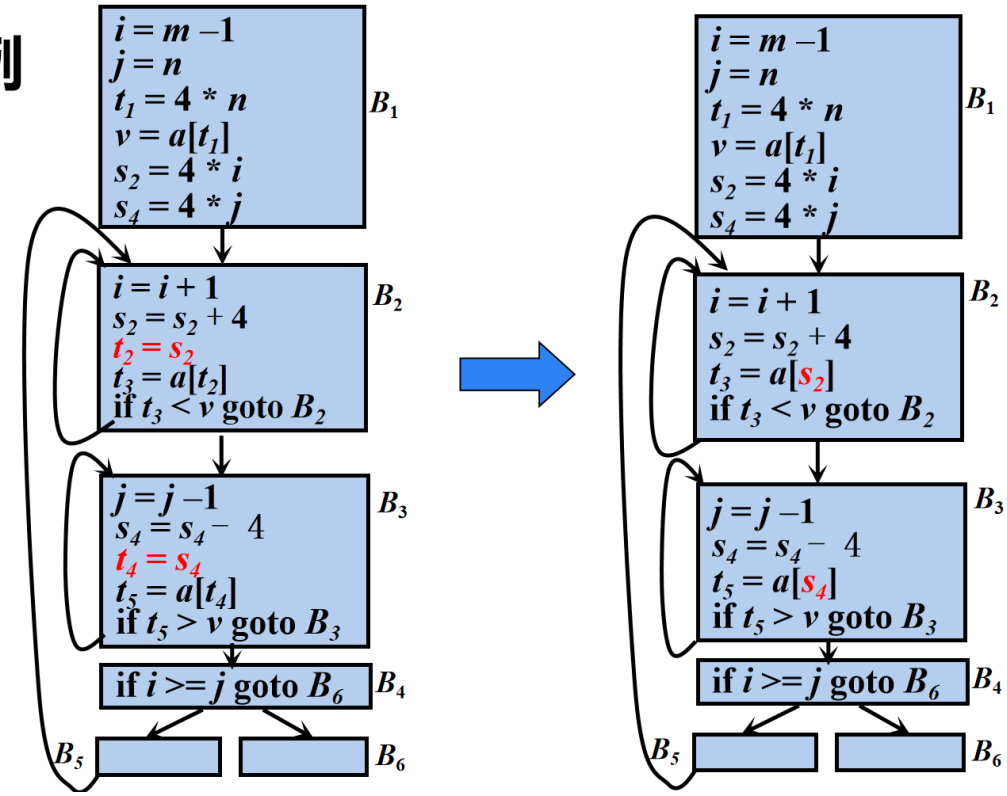
例



⑤ 归纳变量的删除

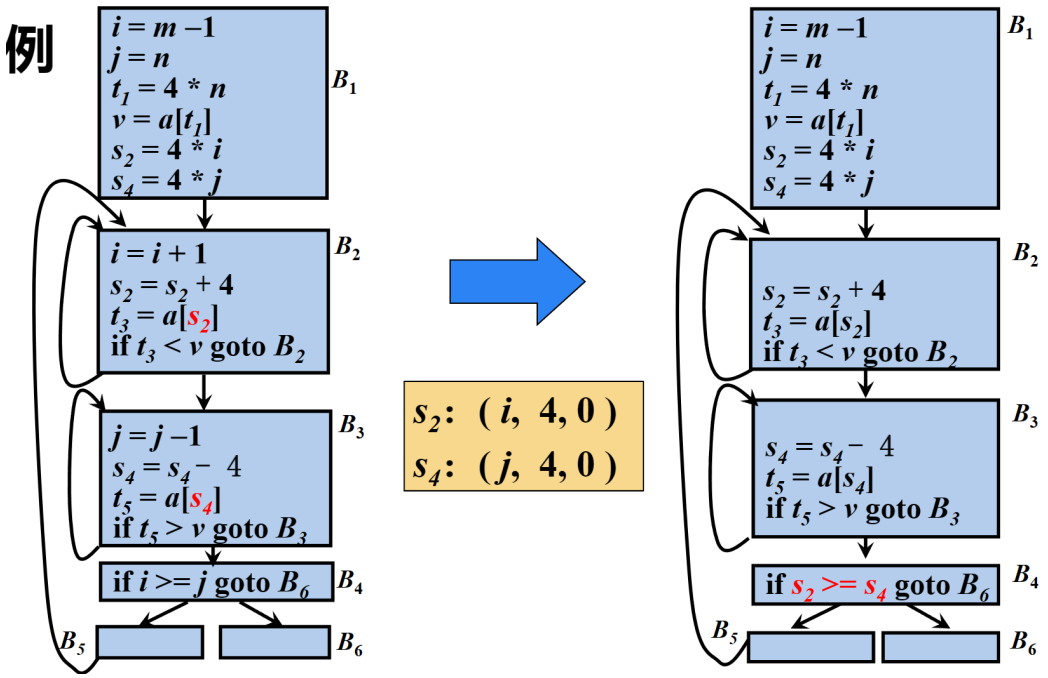
➢ 对于在强度削弱算法中引入的复制语句j=t，如果在归纳变量j的所有引用点都可以用对t的引用代替对j的引用，并且j在循环的出口处不活跃，则可以删除复制语句j=t

例



- 强度削弱后，有些归纳变量的作用只是用于测试。如果可以用对其它归纳变量的测试代替对这种归纳变量的测试，那么可以删除这种归纳变量

- 对于仅用于测试的基本归纳变量i，取i族的某个归纳变量j(尽量使得c、d简单，即c=1或d=0的情况)。把每个对i的测试替换成为对j的测试
  - (*relop* i x B)替换为(*relop* j $c*x+d$ B)，其中x不是归纳变量，并假设$c>0$
  - (*relop* i_1 i_2 B)，如果能够找到三元组j_1(i_1, c, d)和j_2(i_2, c, d)，那么可以将其替换为(*relop* j_1 j_2 B) (假设$c>0$)。否则，测试的替换可能是没有价值的
- 如果归纳变量i不再被引用，那么可以删除和它相关的指令



第九章 代码生成

9.1 代码生成器的主要任务

- 指令选择
  - 选择适当的目标机指令来实现中间表示(IR)语句

- 例：
  - 三地址语句
    - x = y + z
  - 目标代码
    - LD R0, y /* 把y的值加载到寄存器R0中 */
    - ADD R0, R0 , z /* z加到R0上 */
    - ST x , R0 /* 把R0的值保存到x中 */

- 例：

三地址语句序列
a=b+c
d=a+e

目标代码	
LD R0, b	// R0 = b
ADD R0, R0, c	// R0 = R0 + c
ST a , R0	// a = R0
LD R0, a	// R0 = a
ADD R0, R0, e	// R0 = R0 + e
ST d , R0	// d = R0

冗余操作

- 寄存器分配（allocation）和指派（assignment）
  - 把哪个值放在哪个寄存器中
- 指令排序
  - 按照什么顺序来安排指令的执行

9.2 一个简单的目标机模型

- 三地址机器模型
  - 加载、保存、运算、跳转等操作
  - 内存按字节寻址
  - *n*个通用寄存器*R0, R1, …, Rn-1*
  - 假设所有的运算分量都是整数
  - 指令之间可能有一个标号 跳转指令

目标机器的主要指令

- 加载指令 *LD dst, addr*
  - *LD r, x*
  - *LD r_1, r_2*
- 保存指令 *ST x, r*
- 运算指令 *OP dst, src1, src2*
- 无条件跳转指令 *BR L*
- 条件跳转指令 *Bcond r, L*
  - 例: *BLTZ r, L*

寻址模式

- 变量名*a*
  - 例: *LD R1 , a*
    - *R1 = contents (a)*
- *a(r)*
  - *a*是一个变量，*r*是一个寄存器
  - 这个寻址方式对于数组访问是很有用的
    - 其中，*a*是数组的基地址，*r*中存放了数组元素的偏移地址
  - 例: *LD R1 , a(R2)*
    - *R1 = contents (a + contents(R2))*

- *c(r)*
  - c是一个整数，寄存器r 中存放的是一个地址
  - *c(r)*所表示的内存地址是寄存器*r*中的值加上整数*c*
  - 这个寻址方式可以用于沿指针取值
  - 例：*LD R1 , 100 (R2)*
    - *R1 = contents (contents(R2) + 100)*

- **r*
  - 在寄存器*r*的内容所表示的位置上存放的内存位置
  - 例：*LD R1 , * R2*
    - *R1 = conents (contents (contents (R2)))*

- **c(r)*
  - 在寄存器*r*中内容加上*c*后所表示的位置上存放的内存位置
  - 例：*LD R1 , *100(R2)*
    - *R1 = conents (contents (100 + contents(R2)))*

- *#c*
  - 例：*LD R1 , #100*
    - *R1 = 100*

contents(x)表示x所代表的寄存器或内存位置中存放的内容

地址

	例	效果
a	LD R1 , a	R1 = contents (a)
r	LD R1 , R2	R1 = contents (R2)
a(r)	LD R1 , a(R2)	R1 = contents (a + contents(R2))
c(r)	LD R1 , 100 (R2)	R1 = contents (100 + contents(R2))
*c(r)	LD R1 , *100(R2)	R1 = contents (contents (100 +contents(R2)))
*r	LD R1 , * R2	R1 = contents (contents (contents (R2)))
#c	LD R1 , #100	R1 = 100

ST 0(SP), #here + 16

BR *0(SP)

9.3 指令选择

数组寻址语句的目标代码

- 三地址语句
  - b = a[i]
  - a是一个实数数组，每个实数占8个字节
- 目标代码
  - LD R1 , i // R1 = i
  - LD R2 , a(R1) // R2=contents (a + contents(R1))
  - ST b , R2 // b = R2

寻址模式

- 变量名a
- a(r)
- c(r)
- *r
- *c(r)
- #c

- 三地址语句
  - a [j] = c
  - a是一个实数数组，每个实数占8个字节
- 目标代码
  - LD　R1 ,　c　　// R1 = c
  - LD　R2 ,　j　　// R2 = j
  - ST　a(R2) , R1　　// contents(a+contents(R2))=R1

指针存取语句的目标代码

- 三地址语句
  - x = *p
- 目标代码
  - LD R1, p　　// R1 = p
  - LD R2, 0 (R1)　　// R2 = contents (0 + contents (R1))
  - ST x , R2　　// x = R2

寻址模式
- 变量名a
- a(r)
- c(r)
- *r
- *c(r)
- #c

- 三地址语句
  - *p = y
- 目标代码
  - LD　R1 ,　p　　// R1 = p
  - LD　R2 ,　y　　// R2 = y
  - ST　0(R1), R2　　//contents (0 + contents (R1)) = R2

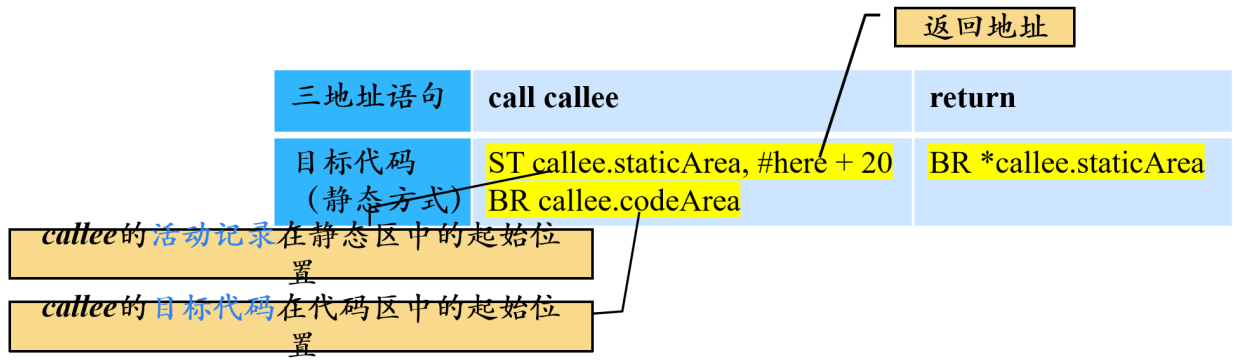
条件跳转语句的目标代码

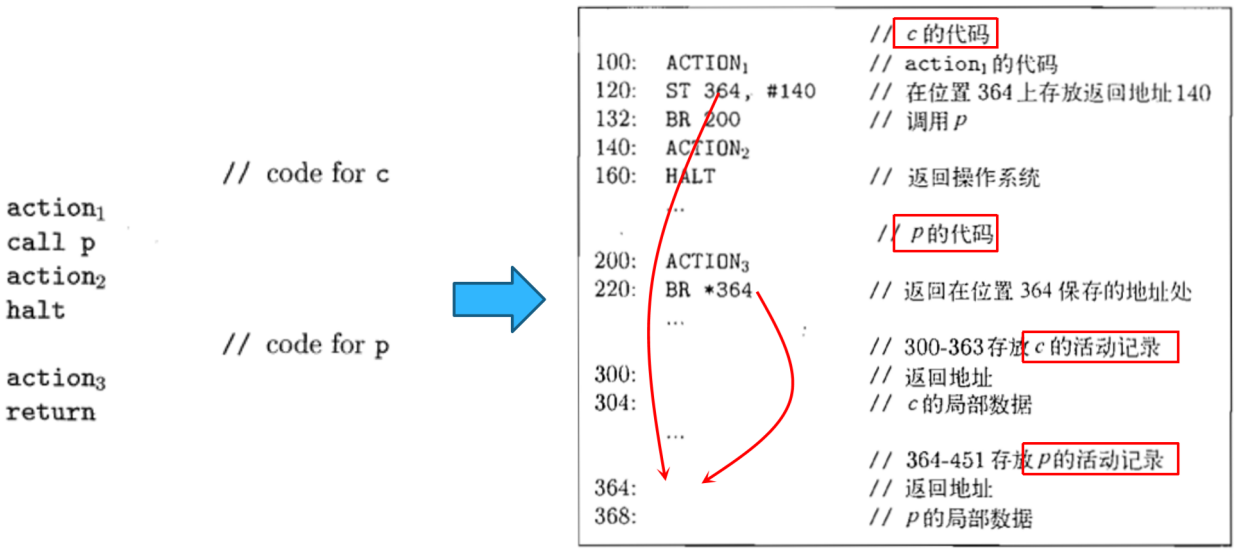
- 三地址语句
  - *if x < y goto L*
- 目标代码
  - *LD R1 , x* // *R1 = x*
  - *LD R2 , y* // *R2 = y*
  - *SUB R1 , R1 , R2* // *R1=R1 - R2*
  - *BLTZ R1 , M* // *if R1 < 0 jump to M*

> *M*是标号为*L*的三地址指令所产生的目标代码中的第一个指令的标号

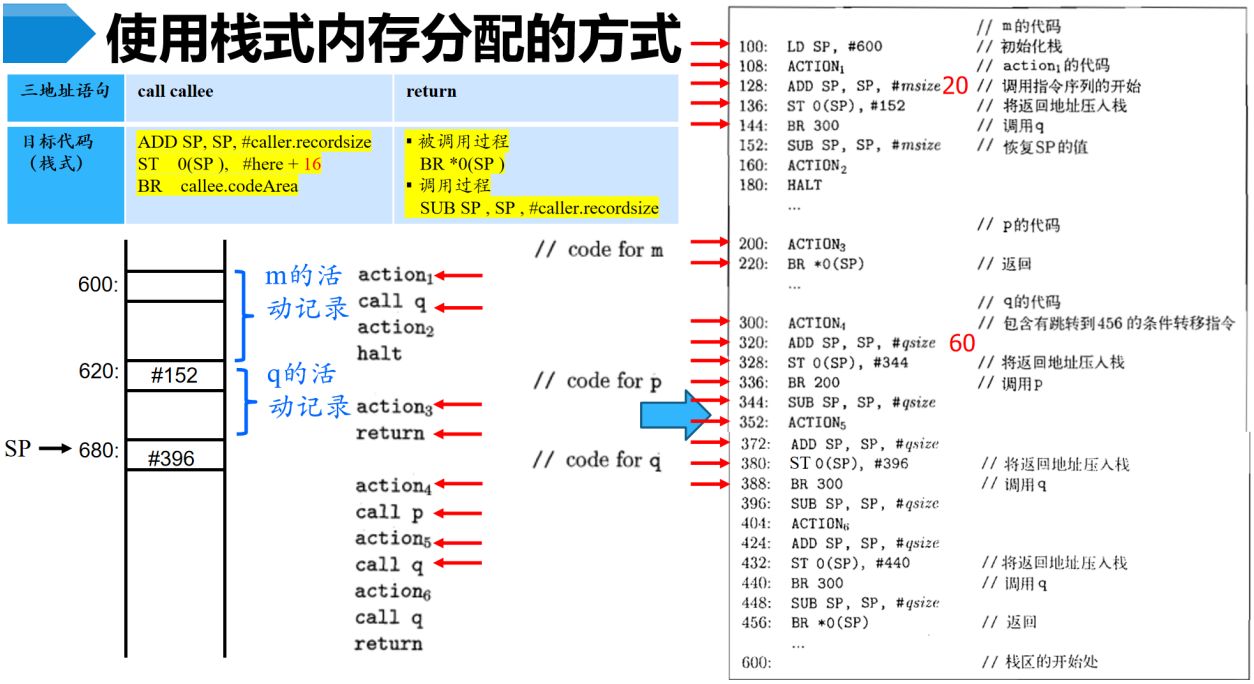
过程调用和返回的目标代码

使用静态内存分配的方式





使用栈式内存分配的方式



P27

9.4 寄存器的选择

三地址语句的目标代码生成

- 对每个形如$x = y\ \mathrm{op}\ z$的三地址指令I，执行如下动作
  - 调用函数$getReg(I)$来为x、y、z选择寄存器，把这些寄存器称为R_x、R_y、R_z
  - 如果R_y中存放的不是y，则生成指令“$LD\ R_y, y'$”。y'是存放y的内存位置之一
  - 类似的，如果R_z中存放的不是z，生成指令“$LD\ R_z, z'$”
  - 生成目标指令“$OP\ R_x, R_y, R_z$”

寄存器描述符和地址描述符

- 寄存器描述符（*register descriptor*）
  - 记录每个寄存器当前存放的是哪些变量的值
- 地址描述符（*address descriptor*）
  - 记录运行时每个名字的当前值存放在哪个或哪些位置
  - 该位置可能是寄存器、栈单元、内存地址或者是它们的某个集合
  - 这些信息可以存放在该变量名对应的符号表条目中

基本块的收尾处理

- 在基本块结束之前，基本块中使用的变量可能仅存放在某个寄存器中
- 如果这个变量是一个只在基本块内部使用的临时变量，当基本块结束时，可以忘记这些临时变量的值并假设这些寄存器是空的
- 对于一个在基本块的出口处可能活跃的变量x，如果它的地址描述符表明它的值没有存放在x的内存位置上，则生成指令“$ST\ x, R$”（R是在基本块结尾处存放x值的寄存器）

管理寄存器和地址描述符

- 当代码生成算法生成加载、保存和其他指令时，它必须同时更新寄存器和地址描述符
  - 对于指令“$LD\ \ R, x$”
    - 修改R的寄存器描述符，使之只包含x
    - 修改x的地址描述符，把R作为新增位置加入到x的位置集合中
    - 从任何不同于x的地址描述符中删除R
  - 对于指令“$OP\ R_x, R_y, R_z$”
    - 修改R_x的寄存器描述符，使之只包含x
    - 从任何不同于R_x的寄存器描述符中删除x
    - 修改x的地址描述符，使之只包含位置R_x
    - 从任何不同于x的地址描述符中删除R_x
  - 对于指令“$ST\ \ x, R$”
    - 修改x的地址描述符，使之包含自己的内存位置

- 对于复制语句x=y，如果需要生成加载指令“LD R_y，y'”则
  - 修改R_y的寄存器描述符，使之只包含y
  - 修改y的地址描述符，把R_y作为新增位置加入到y的位置集合中
  - 从任何不同于y的变量的地址描述符中删除R_y
  - 修改R_y的寄存器描述符，使之也包含x
  - 修改x的地址描述符，使之只包含R_y

例

$t = a - b$
$u = a - c$
$v = t + u$
$a = d$
$d = v + u$

```
LD  R1 , a
LD  R2 , b
SUB R2 , R1 , R2
```

R1	R2	R3
a	t	

a	b	c	d	t	u	v
a , R1	b	c	d	R2		

P41

$t = a - b$
$u = a - c$
$v = t + u$
$a = d$
$d = v + u$
exit

```
ST  a , R2
ST  d , R1
```

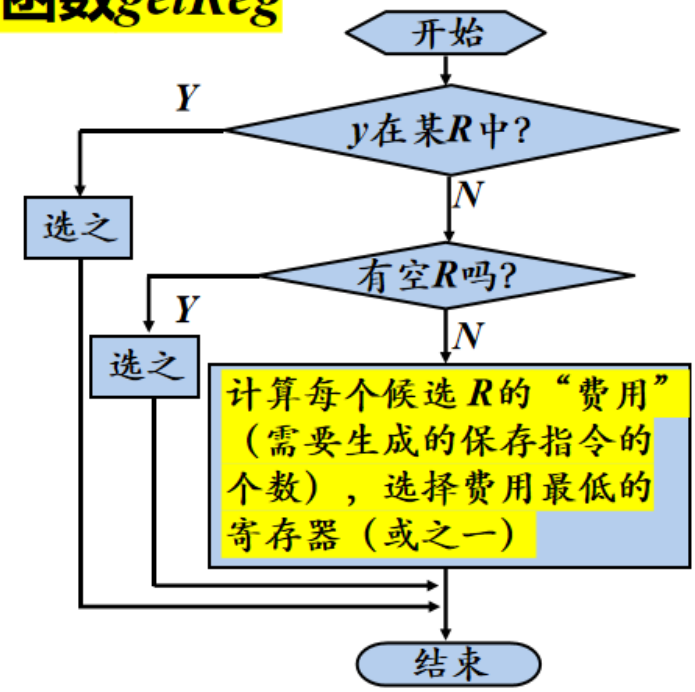
R1	R2	R3
d	a	v

a	b	c	d	t	u	v
R2 ,a	b	c	R1 ,d			R3

寄存器选择函数*getReg*

$x = y\ \mathrm{op}\ z$

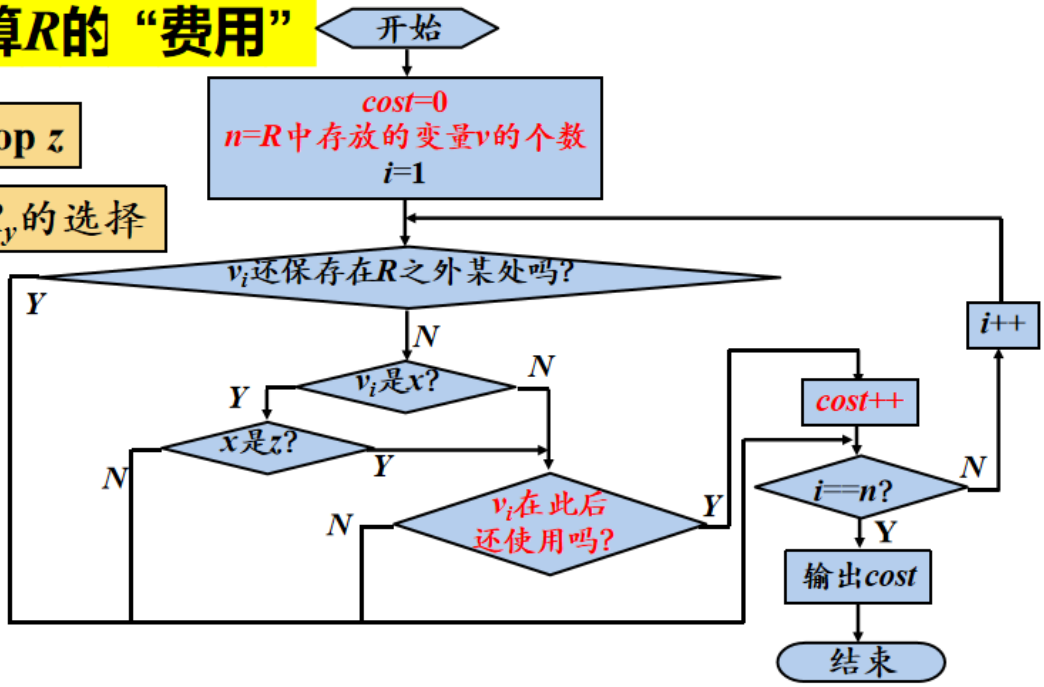
寄存器R_y的选择



计算*R*的“费用”

$x = y\ \mathrm{op}\ z$

寄存器R_y的选择



寄存器R_x的选择 $x = y \text{ op } z$

- 选择方法与R_y类似，区别之处在于
  - 因为x的一个新值正在被计算，因此只存放了x的值的寄存器对R_x来说总是可接受的，即使x就是y或z之一(因为我们的机器指令允许一个指令中的两个寄存器相同)
  - 如果y在指令I之后不再使用，且(在必要时加载y之后) R_y仅仅保存了y的值，那么，R_y同时也可以用作R_x。对z和R_z也有类似选择

    当I是复制指令$x=y$时，选择好R_y后，令$R_x = R_y$

9.5 窥孔优化

- 窥孔(*peephole*)是程序上的一个小的滑动窗口
- 窥孔优化是指在优化的时候，检查目标指令的一个滑动窗口(即窥孔)，并且只要有可能就在窥孔内用更快或更短的指令来替换窗口中的指令序列
- 也可以在中间代码生成之后直接应用窥孔优化来提高中间表示形式的质量

具有窥孔优化特点的程序变换的例子

冗余指令删除

- 消除冗余的加载和保存指令
  - 例

三地址指令序列
- a=b+c
- d=a+e

目标代码
- LD R0，b // R0 = b
- ADD R0，R0，c // R0 = R0 + c
- ST a ，R0 // a = R0
- LD R0， a // R0 = a
- ADD R0，R0，e // R0 = R0 + e
- ST d ，R0 // d = R0

如果第四条指令有标号，则不可以删除

- 消除不可达代码
  - 一个紧跟在无条件跳转之后的不带标号的指令可以被删除
    - 例

```
    if debug == 1 goto L1
    goto L2
L1: print debugging information
L2:
```

→

```
    if debug != 1 goto L2
    print debugging information
L2:
```

debug=0 ↓

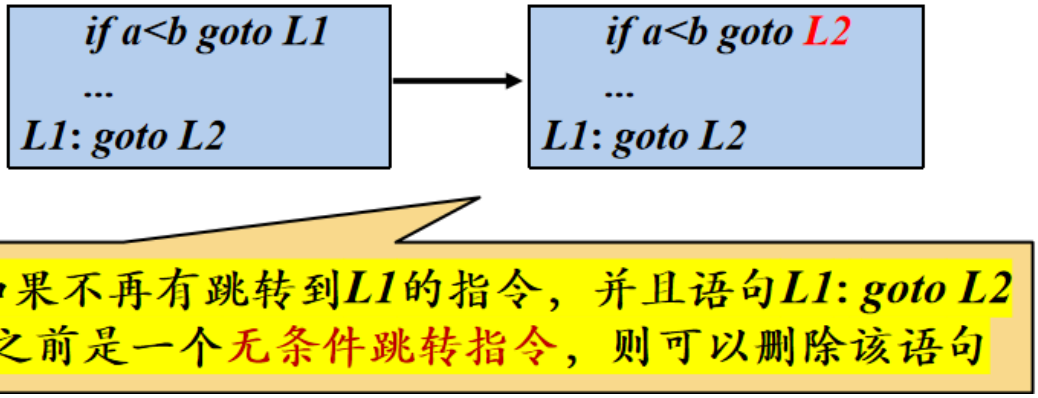
```
    if 0 != 1 goto L2
    print debugging information
L2:
```

←

```
    goto L2
    print debugging information
L2:
```

控制流优化

- 在代码中出现跳转到跳转指令的指令时，某些条件下可以使用一个跳转指令来代替
  - 例



代数优化

- 代数恒等式
  - 消除窥孔中类似于$x=x+0$或$x=x*1$的运算指令
- 强度削弱
  - 对于乘数(除数)是2的幂的定点数乘法(除法)，用移位运算实现代价比较低
  - 除数为常量的浮点数除法可以通过乘数为该常量倒数的乘法来求近似值

特殊指令的使用

- 充分利用目标系统的某些高效的特殊指令来提高代码效率
  - 例如：*INC*指令可以用来替代加1的操作